上海科技大学
ShanghaiTech University

CS289: **Mobile Manipulation Fall 2023**

Sören Schwertfeger

ShanghaiTech University

STAR CENTER

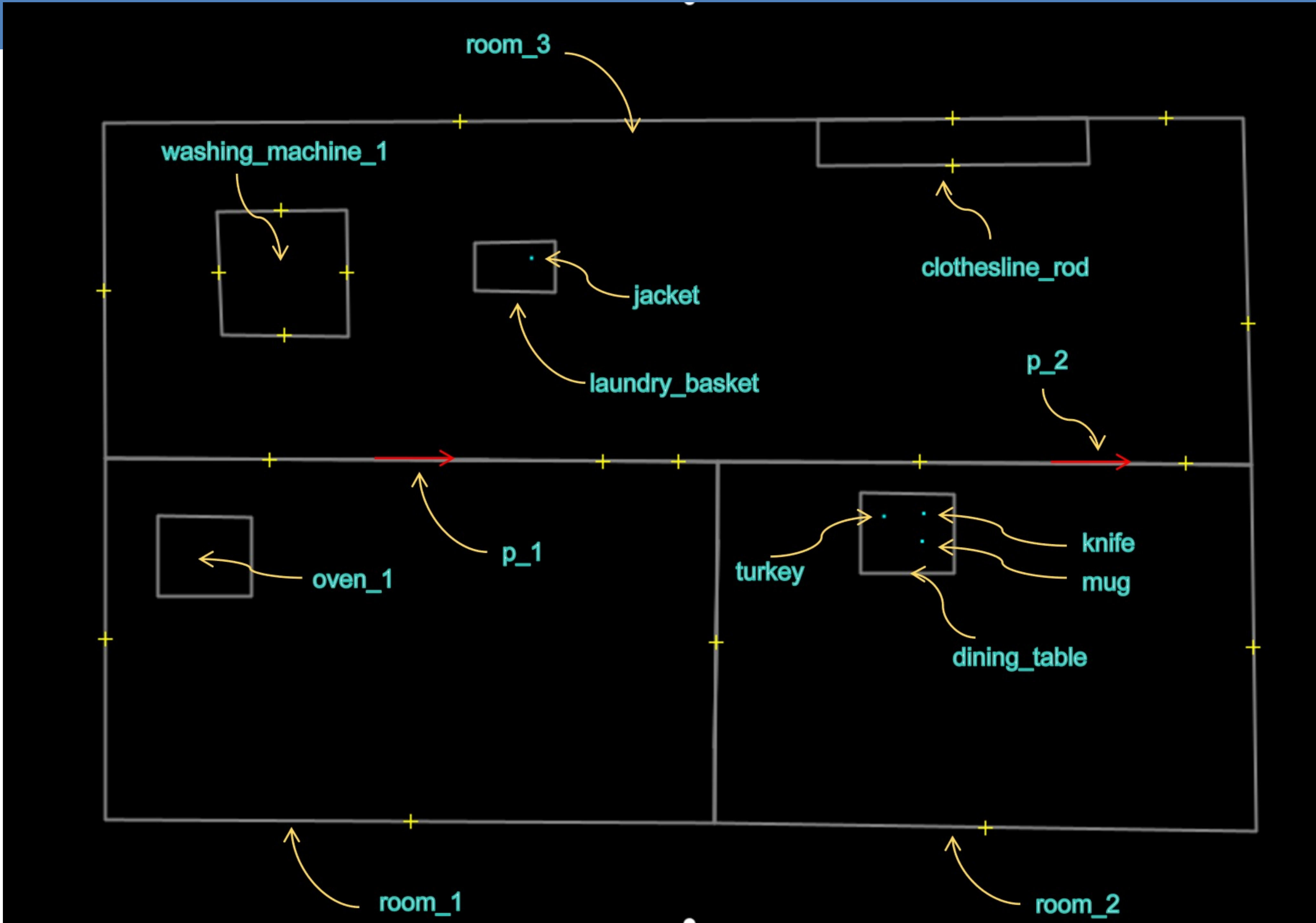
Motivation

- So far in course:
  - Basic capabilities:
    - Arm Planning
    - Mobile base planning & navigation
    - Object recognition
    - Grasping (future lecture)
    - Arm Control (future lecture)
- How to decide what to do when, with which item(s)?
- => Task/ Mission Planning

HW 4

- Execute Task:
- Cook Turkey
- Wash Jacket

- Use:
  - BT
  - State Machine
  - Symbolic Planner
  - LLM



Actions

- Finite set of actions that
- Change the state of the world & robot
- Most often actions have pre-conditions

- Actions are executed using established robot capabilities/ algorithms

- E.g.:
  - Move robot from A to B: A* Path planning & Navigation, e.g. with ROS move_base
  - Pick object: Object detection; arm IK & FK & 3D sensor &
    octree for RRT Planning; Robot control via PID: e.g. with ROS MoveIt
  - Open door: learned how to do it with simulation and RL

Executing Actions in ROS

- Option 1: Just send a ROS message
  - Bad: no explicit feedback – need to program feedback and success by hand
- Option 2: Use ROS service
  - Request and Response pair
  - Caller (can) block till the response is there
  - Useful for very fast actions (e.g. “take picture”)
  - https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html
- Option 3: Use Action
  - Most action take time! E.g. move robot, move robot arm, plan a trajectory, …
  - Actions are asynchronous
  - We (can) get continuous feedback on the operation
  - https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

Mission Execution

Reactive

- Based on current state
- Decide on next action (only)
- Without (explicitly) searching different future options
- Intelligence in the code/ structure, added by programmer who understands the problem

- By hand coding
- **State Machine, Behavior Tree**

- Reinforcement Learning

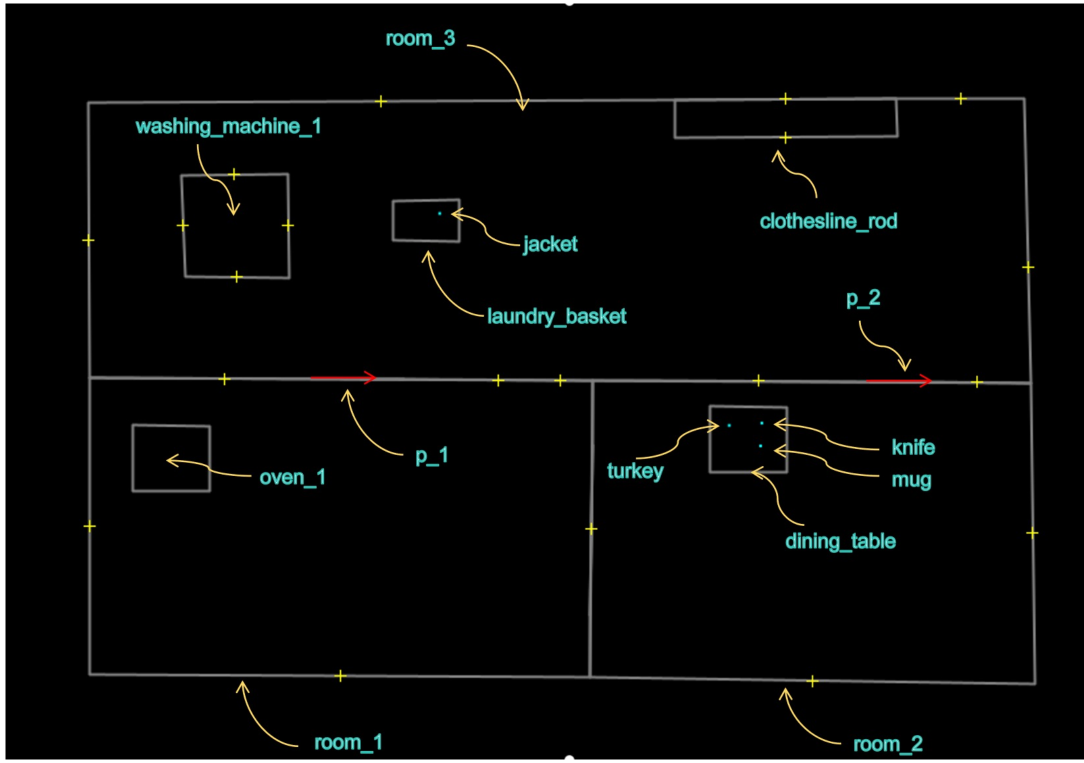
Planning

- Based on the current state
- Look into the future
- Create list of actions to execute till goal
- Often search in space of states, modified by actions

- **Planning Domain Definition Language PDDL**
- Search-based planners, e.g. STRIPS
- With uncertainty:
  - Markov Decision Process (MDP) Planner &
  - Partially Observable MDP (POMDP) Planner
- **LLM** ?
- Combined Task & Motion Planner

HW 4

- Execute Task: Cook Turkey & Wash Jacket
- We provide: simulator that executes 10 simple actions:
  - close_door, open_door; close_furniture; open_furniture; move_robot; pick; place; wash; cook; hang_clothes
- Map is fixed
- Robot start room and initial turkey location are randomized
- Robot is clumsy: door opening has failure rate of 50%!
- Goal: execute actions (via ROS Service) such that goal state is reached
- Use: BT; State Machine; Symbolic Planner; LLM
  - Work in groups of 4 – one approach per student!



FINITE STATE MACHINES

With material from
Marco Della Vedova and Tullio Facchinetti
University of Pavia
https://robot.unipv.it/toolleeo/

Motivation

- How should the robot know what to do now?
- Different states, e.g.:
  - Waiting for a task
  - Charging batteries
    - Go to charger
    - Park in charger
  - Executing certain task, pick and place a bottle
    - Go to for picking up bottle
    - Search for bottle
    - Take bottle
    - Go to goal
    - Place bottle
  - Error and recovery states!

Models of computation and abstract machines

In computer science, automata theory studies mathematical objects called *abstract machines*, or *automata*, and the computational problems that they can solve.

Automata comes from the Greek word
αὐτόματα = "self-acting"

An abstract machine (a.k.a. abstract computer) is a theoretical model of a computer hardware or software systems.

A model of computation is the definition of the set of allowable operations used in computation and their respective costs. It is used for:

- measuring the complexity of an algorithm in execution time and/or memory space
- analyze the computational resources required
- software and hardware design

Finite State Machines (FSMs)

- A Finite State Machine (a.k.a. *finite state automaton*) is an abstract device
- It consists of:
  - a set of states (including a start state)
  - an alphabet of symbols that serves as a set of possible inputs to the machine
  - and a transition function that maps each state to another state (or to itself) for any given input symbol
- The machine operates by being fed a string of symbols, and moves through a series of states according to the transition function.
- Output? Different types of FSM are distinguished depending on if the output is produced and how it is produced: before or after a transition.

Example of FSM: an edge-detector

- Detect transitions between two symbols in the input sequence, say 0 and 1.
- Output:
  - 0 if this symbol is the same as the previous symbol
  - 1 if this symbol is different
  - 0 always for first symbol

Examples:

$$\text{inputs} \dashrightarrow \text{outputs}$$

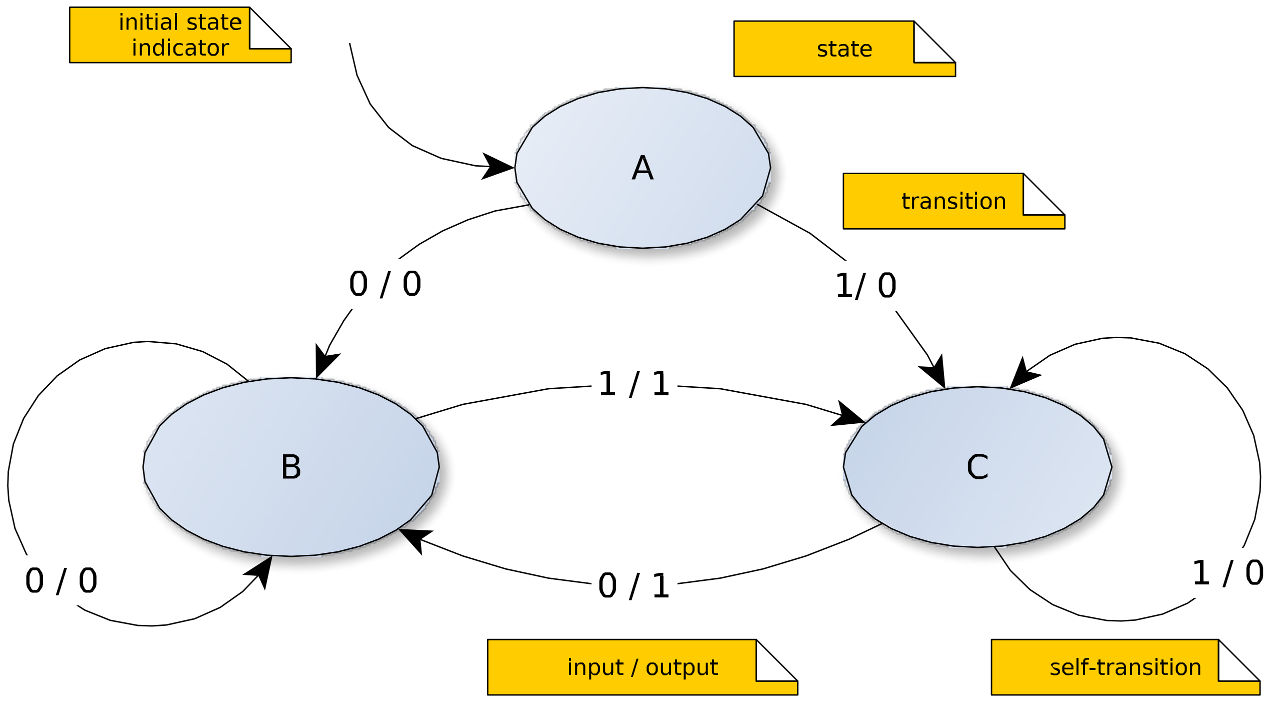
$$0\ 1\ 1\ 1 \dashrightarrow 0\ 1\ 0\ 0$$

$$0\ 1\ 1\ 1\ 1\ 0 \dashrightarrow 0\ 1\ 0\ 0\ 0\ 1$$

$$1\ 0\ 1\ 0\ 1\ 0 \dashrightarrow 0\ 1\ 1\ 1\ 1\ 1$$

Graphical representation of FSM using graphs

Edge-detector example



This graphical representation is known as state diagram.
A state diagram is a direct graph with a special node representing the initial state.

A FSM is a five-tuple

Mathematical model of a FSM

$$(\Sigma, \Gamma, S, s_0, \delta)$$

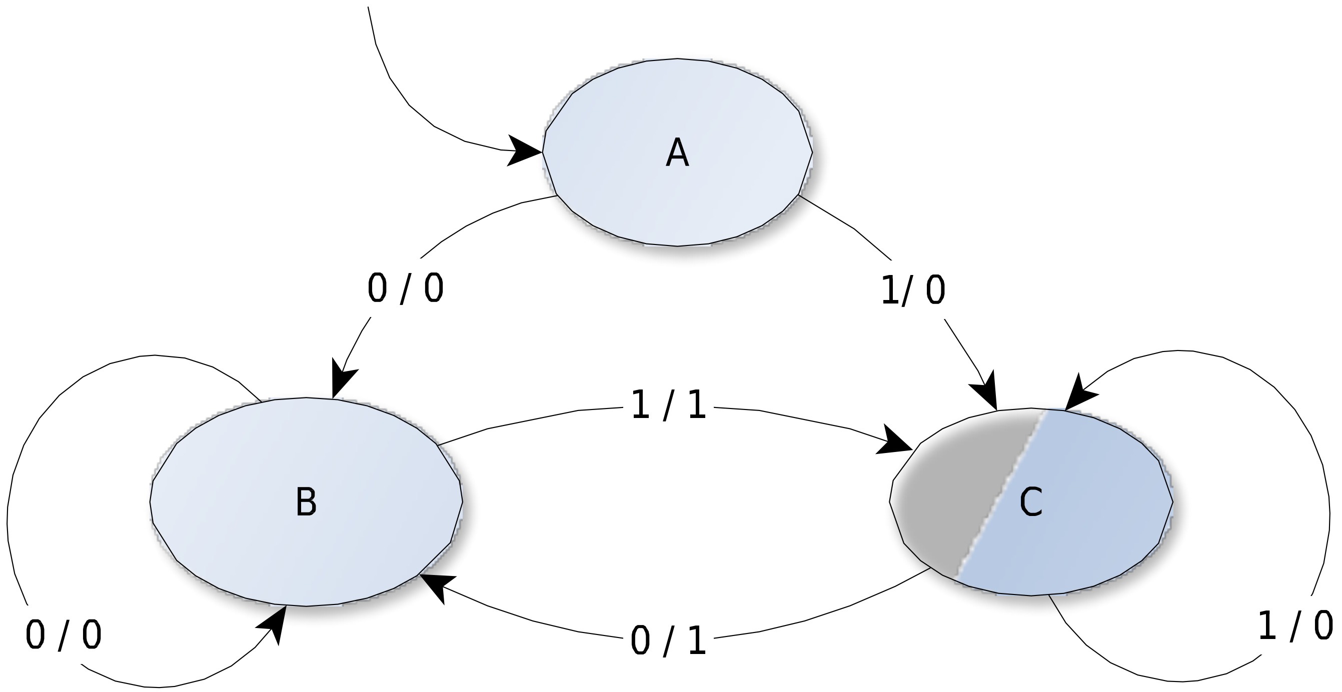
where:

- Σ is the input alphabet (a finite, non-empty set of symbols).
- Γ is the output alphabet (a set of symbols).
- S is a finite, non-empty set of states.
- s_0 is the initial state, an element of S.
- δ is the transition function: $\delta : S \times \Sigma \rightarrow S \times \Gamma$.

Tabular representation of a FSMs' transition function

The transition function $\delta : S \times \Sigma \rightarrow S \times \Gamma$ can be represented by a tabular with states on the rows and inputs on the columns. In each cell there is a tuple s, γ indicating the next state and the output.

For example, for the edge-detector FSM, the transition table is:



	0	**1**
A	B,0	C,0
B	B,0	C,1
C	B,1	C,0

The notion of state

- Intuitively, the state of a system is its condition at a particular point in time
- In general, the state affects how the system reacts to inputs
- Formally, we define the state to be an encoding of everything about the past that has an effect on the system's reaction to current or future inputs

The state is a summary of the past

Transitions

- Transitions between states govern the discrete dynamics of the state machine and the mapping of inputs to outputs. The FSM evolves in a sequence of transitions.

- When does a transition occur?
  - Nothing in the definition of a state machine constrains *when* it reacts.
  - As a discrete system, we do not need to talk explicitly about the amount of time that passes between transitions, since it is actually irrelevant to the behavior of a FSM.

Still, a FSM could be:

- **event triggered** −−→ it reacts whenever an input is provided
- **time triggered** −−→ it reacts at regular time intervals

The definition of the FSM does not change in these two cases. The environment where an FSM operates defines when it should react.

Mealy FSM and Moore FSM

- So far we implicitly dealt with Mealy FSM, named after George Mealy, a Bell Labs engineer who published a description of these machines in 1955.
- Mealy FSM are characterized by producing outputs when a transition is taken.

- An alternative, known as a Moore FSM, produces outputs when the machine is in a state, rather than when transition is taken.
- Moore machines are named after Edward Moore, another Bell Labs engineer who described the model in a 1956 paper.

Mealy FSM and Moore FSM

- The distinction between Mealy and Moore machines is subtle but important.
- Both are discrete systems, and hence their operation consists of a sequence of discrete reactions.
- For a Moore machine, at each reaction, the output produced is defined by the current state (at the start of the reaction, not at the end).
- Thus, the output at the time of a reaction does not depend on the input at that same time.
- The input determines which transition is taken, but not what output is produced by the reaction.

With these assumptions, a Moore machine is strictly causal

Notion of causality

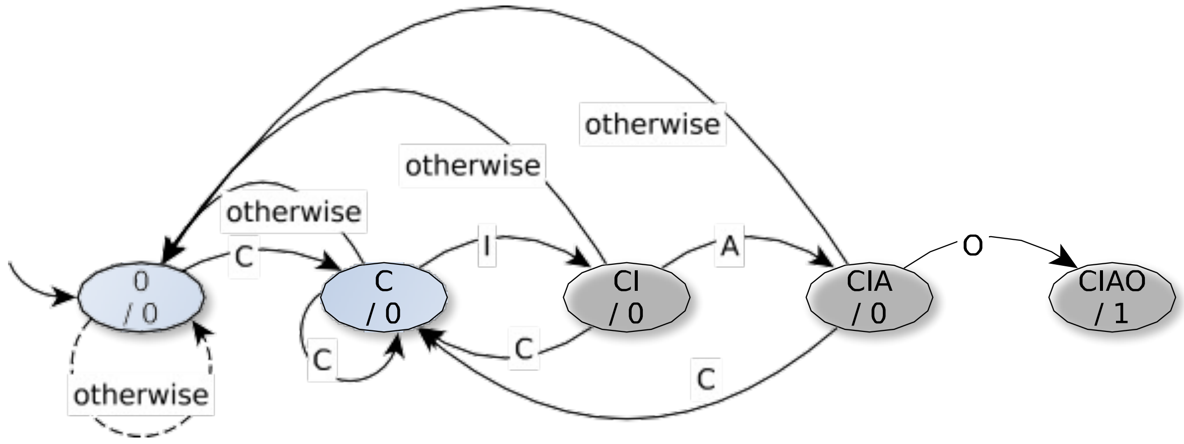
- a system is **causal** if its output depends only on current and past inputs
- in other words, in causal systems if two input sequences are identical up to (and including) time τ, the outputs are identical up to (and including) time τ
- in **strictly causal systems** if two possible inputs are identical up to (and not including) time τ, the outputs are identical up to (and not including) time τ

strictly causal systems are useful to build feedback systems

- **non-causal** (**acausal**) systems depends also on future inputs (examples: population growth, weather forecasting, planning)
- **anti-causal** systems depends only on future inputs

Moore FSM example

Request: Design a Moore FSM that takes characters A-Z as input and returns 1 if in the input there is the string "CIAO".
Note: since the output depends on the current state only, outputs are shown in the state rather than on the transitions in the state diagram.



Notes (valid for Moore and Mealy FSM state diagrams):

- it is often convenient to use the label *otherwise* on transitions
- *otherwise* self-transition are called "default transitions" and can be omitted

Mealy FSM vs Moore FSM

- any Moore machine can be converted to an equivalent Mealy machine
- a Mealy machine can be converted to an almost equivalent Moore machine
- it differs only in that the output is produced on the next reaction rather than on the current one
- Mealy machines tends to be more compact (requiring fewer states to represent the same functionality), and are able to produce an output that instantaneously responds to the input
- Moore machines are used when output is associated with a state of the machine, hence the output is somehow *persistent*

FSM classification

- Transducers are machines that read strings (sequences of symbols taken from an alphabet) and produce strings containing symbols of another (or even the same) alphabet.
- Acceptors (aka recognizers and sequence detectors) produce a binary output, saying either *yes* or *no* to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected.
- Classifiers are a generalization that similarly to acceptors produce a single output when terminates but has more than two terminal states.
- Generators (aka sequencers) are a subclass of aforementioned types that have a single-letter input alphabet. They produce only one sequence, which can be interpreted as output sequence of transducer or classifier outputs.

Extended state machines

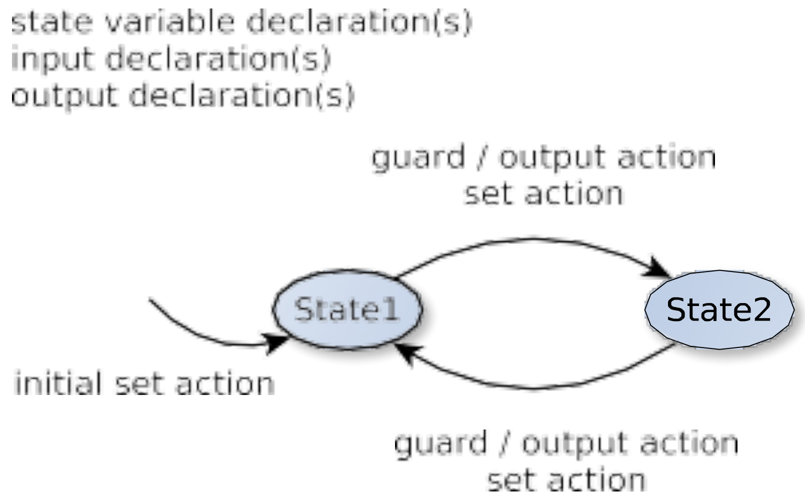
The notation for FSMs becomes awkward when the number of states gets large. Moreover, many applications require to read two or more input sources.

Extended state machines address those issues by augmenting the FSM model

- internal state variables that may be read and written as part of taking a transition between states;
- input valuations: a valuation of a set of variables is an assignment of value to each variable;
- transitions triggered by guards: a guard is a *predicate* (a boolean-valued expression) that evaluates to *true* when the transition should be taken;
- output actions that may be valuations of output variables or function calls.

Extended state machines: graphical notation

The general notation for extended state machines is the following:



- set actions specify assignments to variables that are made when the transition is taken
- these assignments are made *after* the guard has been evaluated and the output actions have been fired
- if there are more than one output action or set action, they are made in sequence

Extended state machine example: traffic light

Problem: model a controller for a traffic light (for cars) at a pedestrian crosswalk.

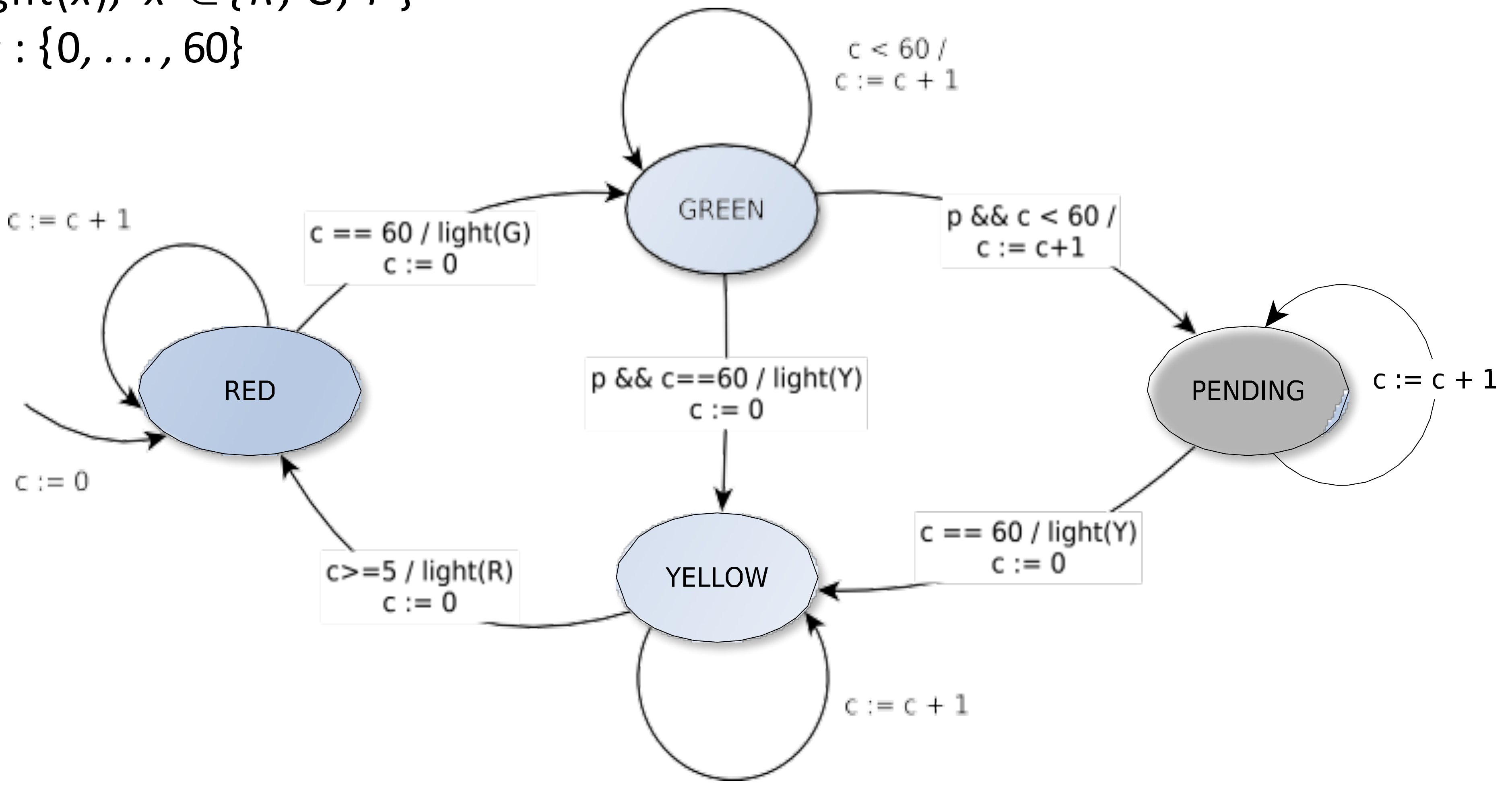
Extended state machine example: traffic light

Problem: model a controller for a traffic light (for cars) at a pedestrian crosswalk.

1. Use a time triggered machine that reacts once per second.
2. It starts in the RED state and counts 60 seconds with the help of the internal variable c.
3. It then transitions to GREEN, where it will remain until the input p is true. That input could be generated by a pedestrian pushing a button to request a walk light.
4. When p is true, the machine transitions to YELLOW if it has been in state GREEN for at least 60 seconds.
5. Otherwise, it transitions to pending, where it stays for the remaining part of the 60 second interval. This ensures that once the light goes green, it stays green for at least 60 seconds.
6. At the end of 60 seconds, it will transition to YELLOW, where it will remain for 5 seconds before transitioning back to RED.
7. The outputs produced by this machine is a function call to $\mathtt{light}(x)$, where $x \in \{R, G, Y\}$ represents the color light to be turned on.

Extended state machine example: traffic light

inputs: $p : \{true, false\}$
outputs: light(x), $x \in \{R, G, Y\}$
variables: $c : \{0, \ldots, 60\}$



Extended state machines: state space

The **state** of an extended state machine includes not only the information about which discrete state the machine is in (indicated by a bubble), but also what values any variables have.
The number of possible states can therefore be quite large, or *even infinite*.
If there are n discrete states (bubbles) and m variables each of which can have one of p possible values, then the size of the state space of the state machine is

$$|\text{States}| = np^m$$

Extended state machines may or may not be FSMs. In particular, it is not uncommon for p to be infinite. For example, a variable may have values in N, the natural numbers, in which case, the number of states is infinite.

Reachable states

Some state machines will have states that can never be reached, so the set of reachable states – comprising all states that can be reached from the initial state on some input sequence – may be smaller than the set of states.

For example, in the traffic light FSM, the c variable has 61 possible values and there are 4 bubbles, so the total number of combination is $61 \times 4 = 244$. The size of the state space is therefore 244.

However, not all of these states are reachable. In particular, while in the YELLOW state, the count variable will have only one of 6 values in $\{0, \ldots, 5\}$.

The number of reachable states, therefore, is $61 \times 3 + 6 = 189$.

Determinacy

- A state machine is said to be **deterministic** (or **determinate**) if, for each state, there is at most one transition enabled by each input value.
- The given formal definition of an FSM ensures that it is deterministic, since the transition function δ is a function, not a one-to-many mapping.
- The graphical notation with guards on the transitions, however, has no such constraint.
- Such a state machine will be deterministic only if the guards leaving each state are *non-overlapping*.

Receptiveness

- A state machine is said to be receptive if, for each state, there is at least one transition possible on each input symbol.
- In other words, receptiveness ensures that a state machine is always ready to react to any input, and does not “get stuck” in any state.
- The formal definition of an FSM given in the previous slides ensures that it is receptive, since δ is a function, not a partial function.
- It is defined for every possible state and input value.
- Moreover, in our graphical notation, since we have implicit default transitions, we have ensured that all state machines specified in our graphical notation are also receptive.

if a state machine is both deterministic and receptive, for every state, there is exactly one transition possible on each input value

Nondeterminism

If for any state of a state machine, there are two distinct transitions with guards that can evaluate to true in the same reaction, then the state machine is nondeterminate or nondeterministic.
It is also possible to define machines where there is more than one initial state: such a state machine is also nondeterminate.

Applications

- modeling unknown aspects of the environment or system
- hiding detail in a specification of the system
- non-deterministic FSMs are more compact than deterministic FSMs
  - a classic result in automata theory shows that a nondeterministic FSM has a related deterministic FSM that is language equivalent
  - but the deterministic machine has, in the worst case, many more states (exponential)

Behaviors, Traces and Computational Trees

- FSM behavior is a sequence of transitions.
- An execution trace is the record of inputs, states, and outputs in a behavior. A trace looks like:

$$((u_0, x_0, y_0), (u_1, x_1, y_1), (u_2, x_2, y_2), \ldots)$$

or

$$x_0 \xrightarrow{u_0/y_0} x_1 \xrightarrow{u_1/y_1} x_2 \xrightarrow{u_2/y_2} \ldots$$

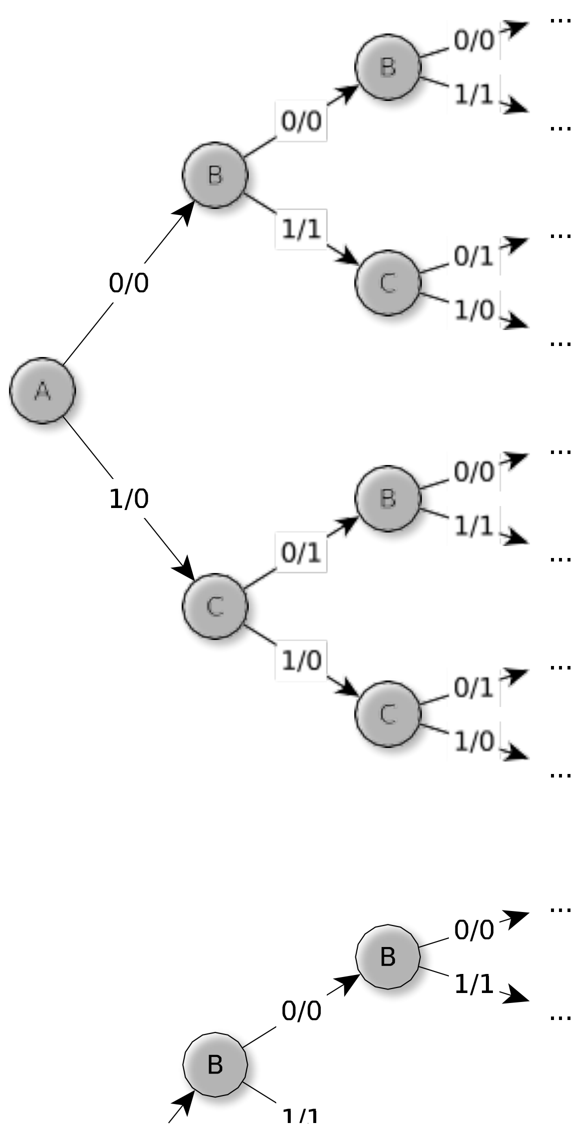
where u_i, x_i, y_i represent valuation of the inputs, current state, and outputs' valuation at transition i, respectively.

- A computational tree is a graphical representation of all possible traces

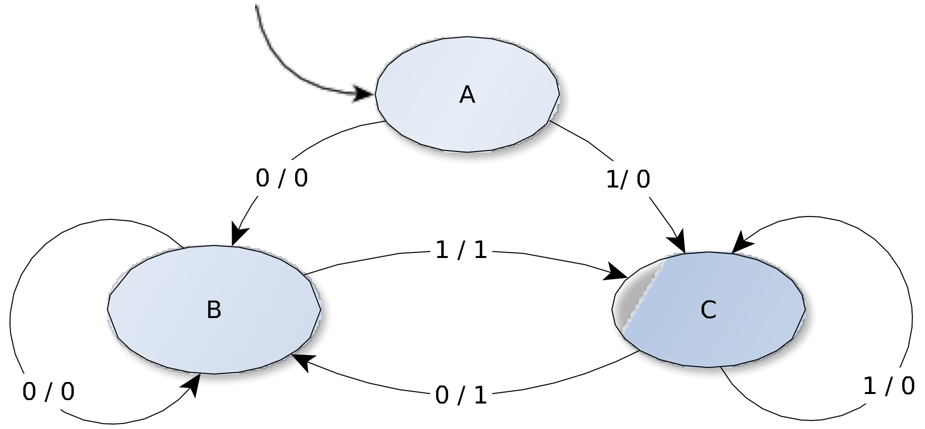
FSMs are suitable for formal analysis. For example, safety analysis might show that some unsafe state is not reachable.

Computational tree example

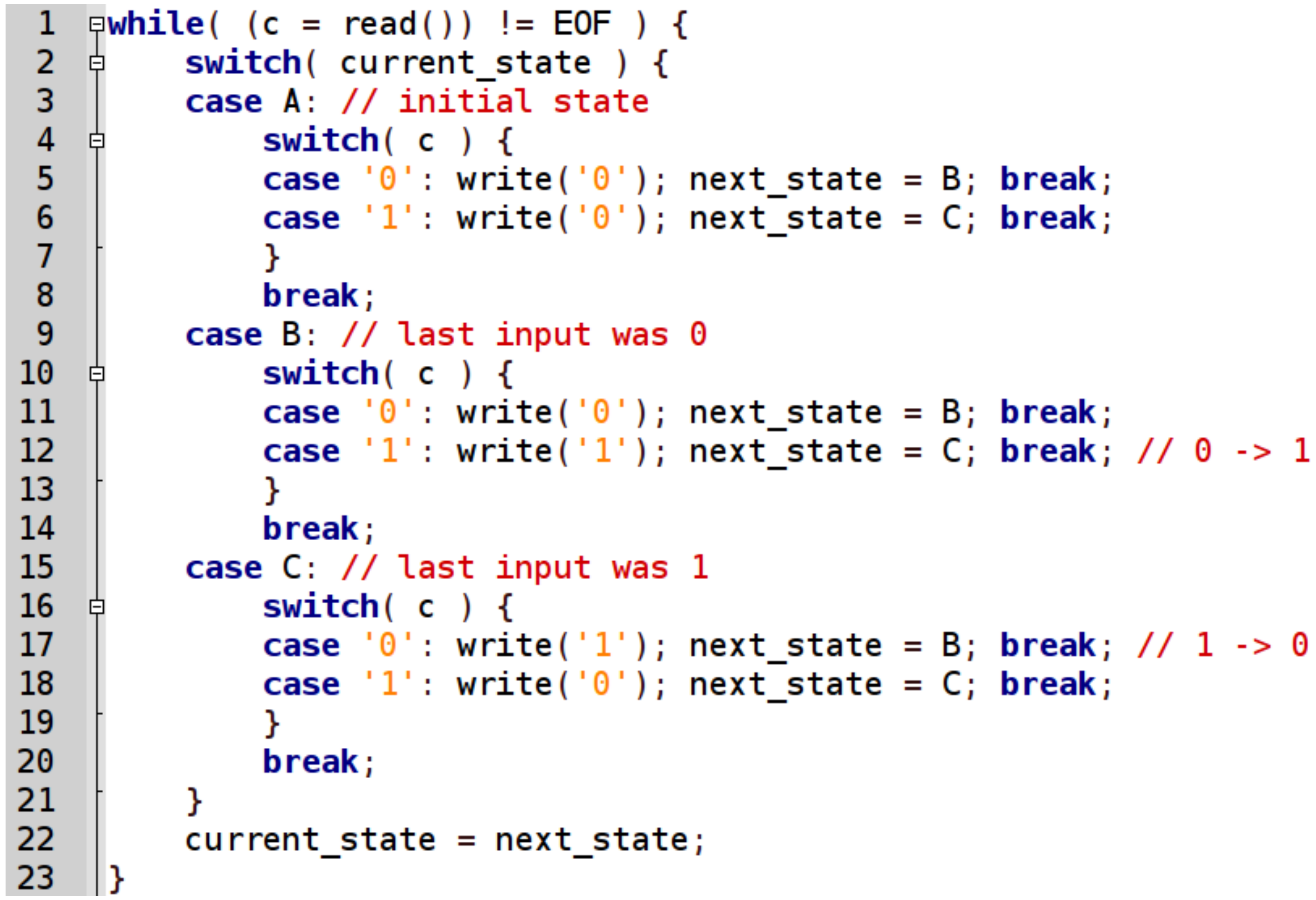
Computational tree:



Recall the edge-detector FSM:



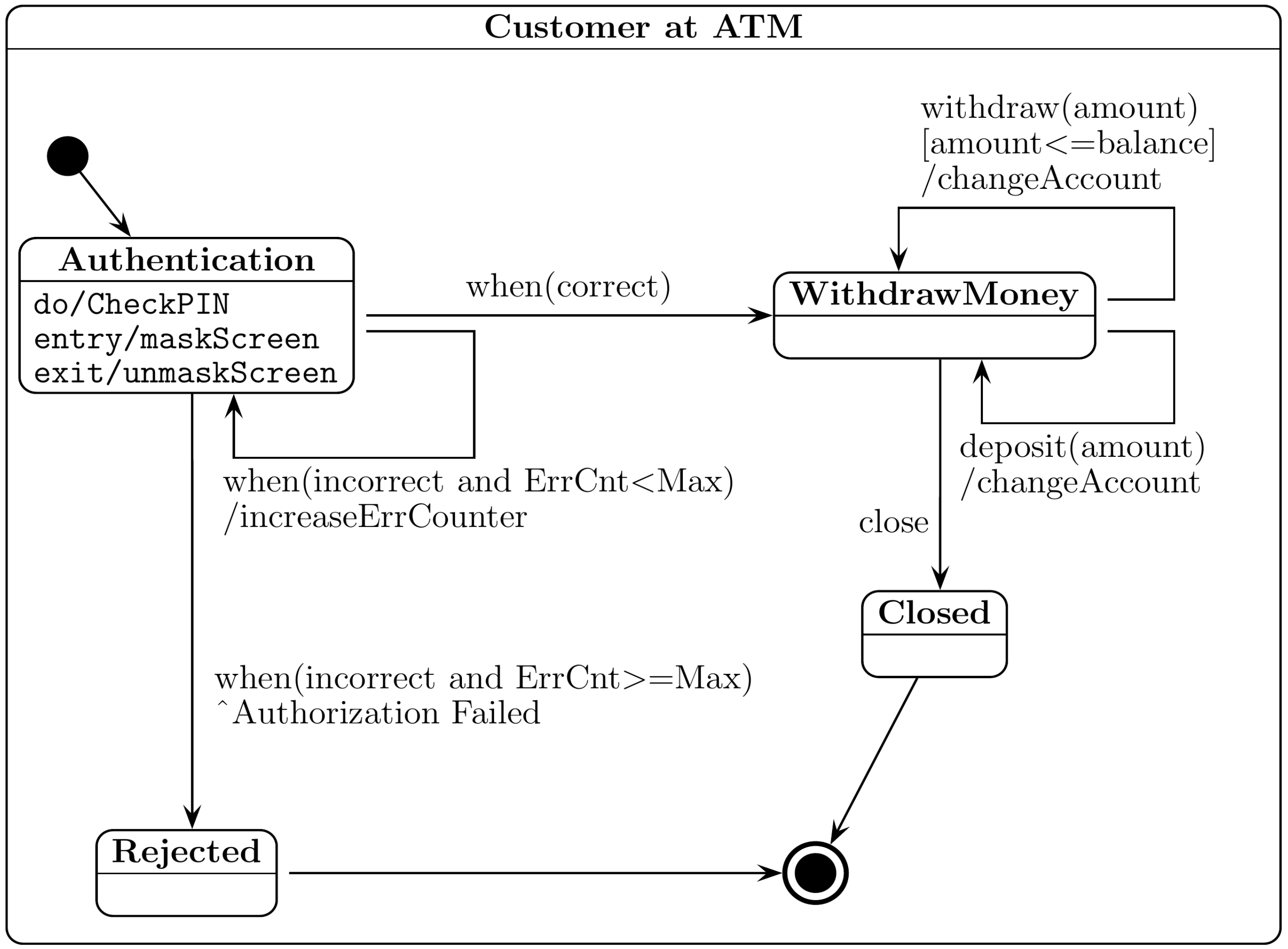
Implementation: imperative programming language



```c
while( (c = read()) != EOF ) {
    switch( current_state ) {
    case A: // initial state
        switch( c ) {
        case '0': write('0'); next_state = B; break;
        case '1': write('0'); next_state = C; break;
        }
        break;
    case B: // last input was 0
        switch( c ) {
        case '0': write('0'); next_state = B; break;
        case '1': write('1'); next_state = C; break; // 0 -> 1
        }
        break;
    case C: // last input was 1
        switch( c ) {
        case '0': write('1'); next_state = B; break; // 1 -> 0
        case '1': write('0'); next_state = C; break;
        }
        break;
    }
    current_state = next_state;
}
```

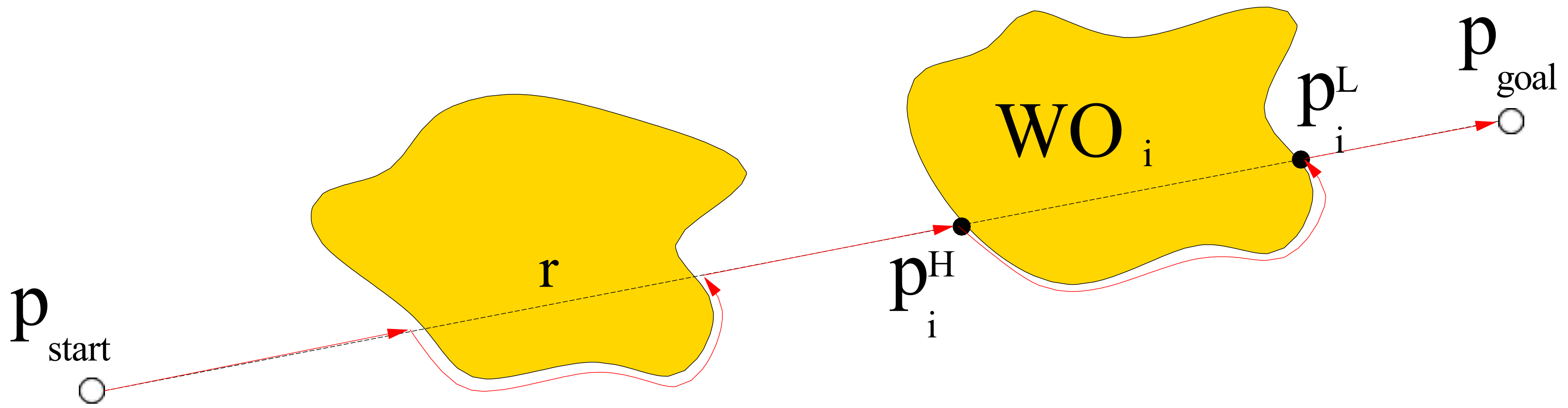
Implementation: UML State Machine Diagram

Example: ATM



Reference: http://www.uml-diagrams.org/state-machine-diagrams.html

Bug 2 - algorithm overview



Essentials:

- motion-to-goal until an obstacle is encountered
- obstacle circumnavigation until the *r* straight line is encountered, i.e., the line connecting the starting point and the goal
- at that point, back to motion-to-goal along the *r* straight line

Bug 2 - hypoteses (1/2)

- Hypoteses:
  - discretized workspace - each point belongs to a finite set *W*
  - **dist(P1,P2)** - a function that computes the distance between P1 and P2
  - **isonR(P1)** - a function that returns true if P1 is on the line *r*
- Input:
  - **touch** - binary variable set by a proximity sensor in front of the robot
  - **pos** - variable in *W*, updated by a position sensor
- Output (actions):
  - **go()** - robot moves along the straight line in front of it
  - **turn(...)** - robot rotates; the action is instantaneous (simplification)
  - **coastObs()** - robot proceeds coasting the obstacle
  - **stop()** - robot stops

Bug 2 - hypoteses (2/2)

- State variables:

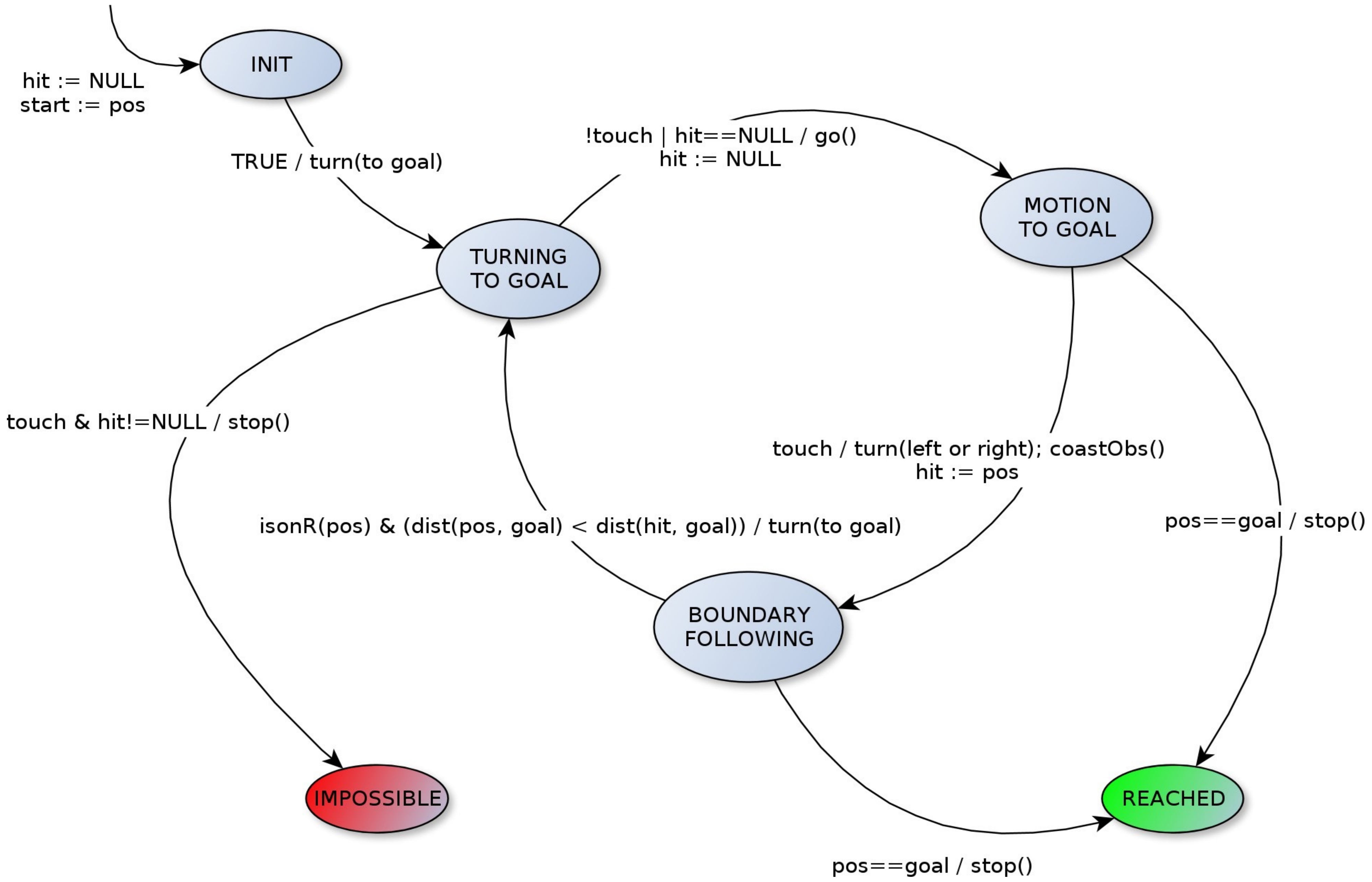
  hit - variable in $W \cup \{NULL\}$, which stores the hit point

  start - variable in W , which stores the starting point. It is necessary for calculating the line start-goal

- Parameter:

  goal - constant in W

Bug 2 Mealy FSM

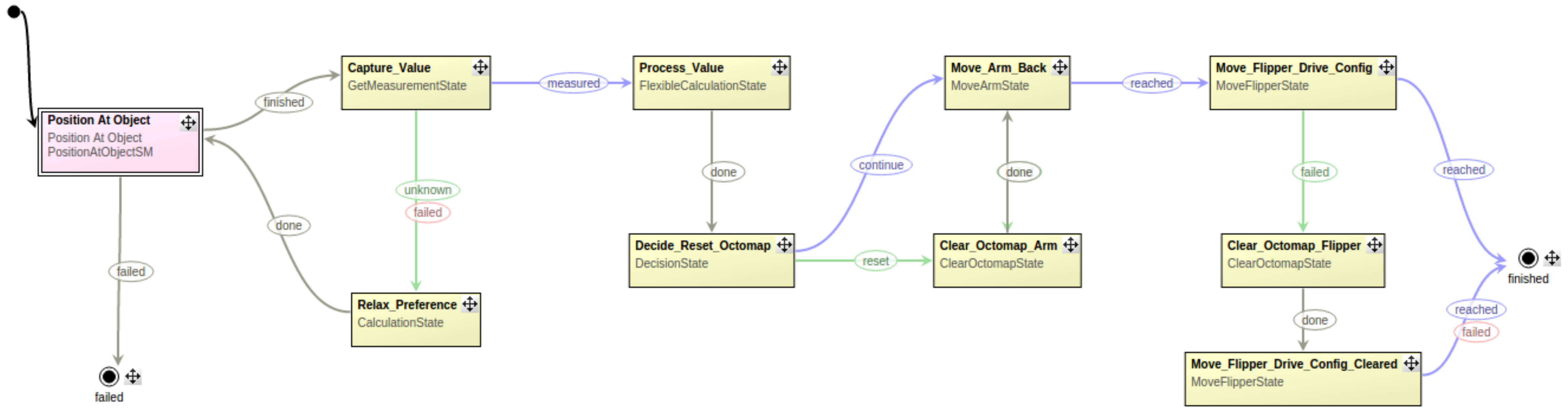


ROS: FLEXBE

FlexBe

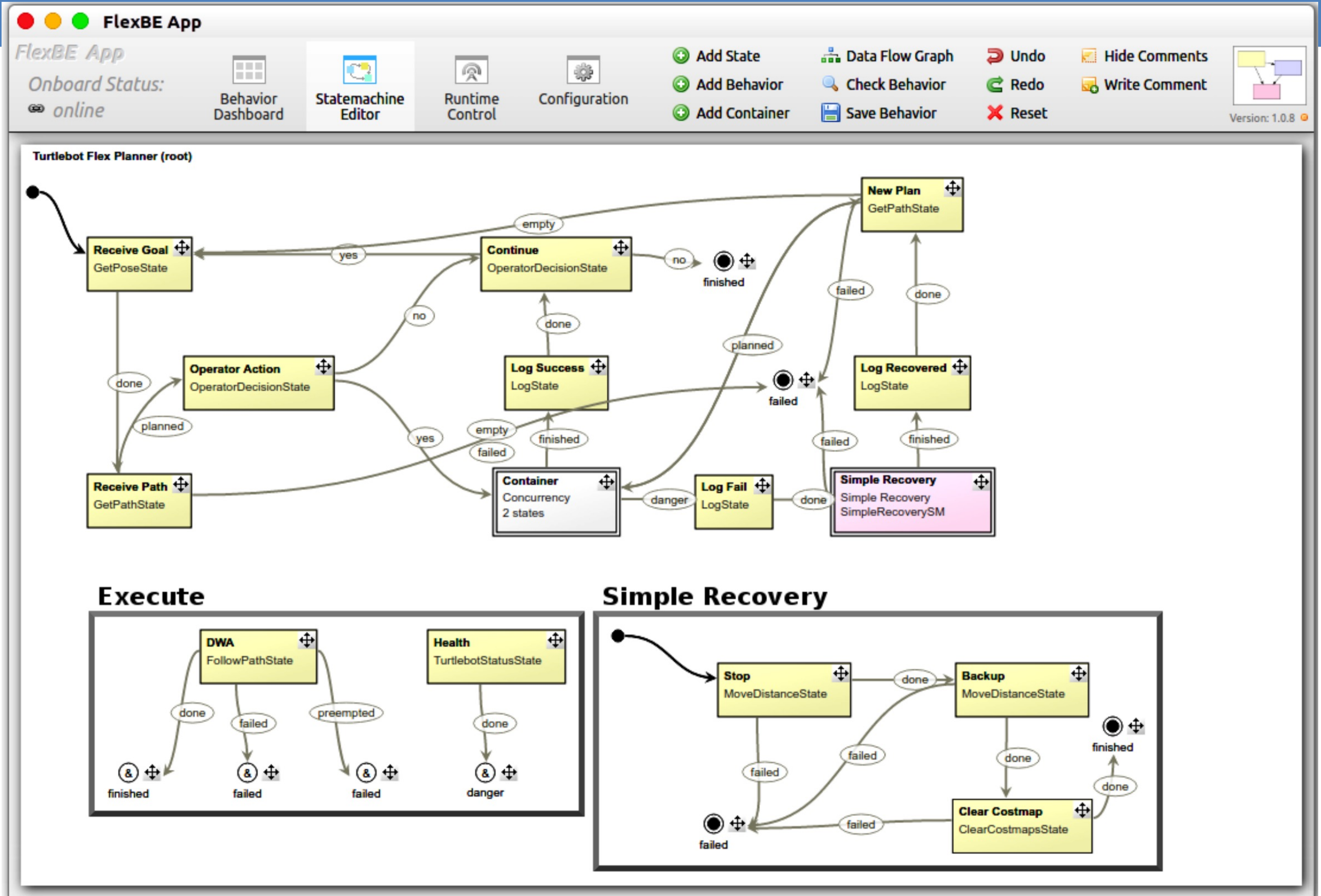
- FlexBE is a powerful and user-friendly high-level behavior engine, flexibly applicable to numerous systems and scenarios.
- drag&drop behavior creation
- automated code generation
- monitor behavior execution
- adjustable level of autonomy
- runtime-modification of behaviors

- Programmable with python for the states



Paper:
Flexible Navigation: Finite State Machine-Based Integrated Navigation and Control for ROS Enabled Robots
https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7925266

http://wiki.ros.org/flexbe https://flexbe.readthedocs.io/en/latest/

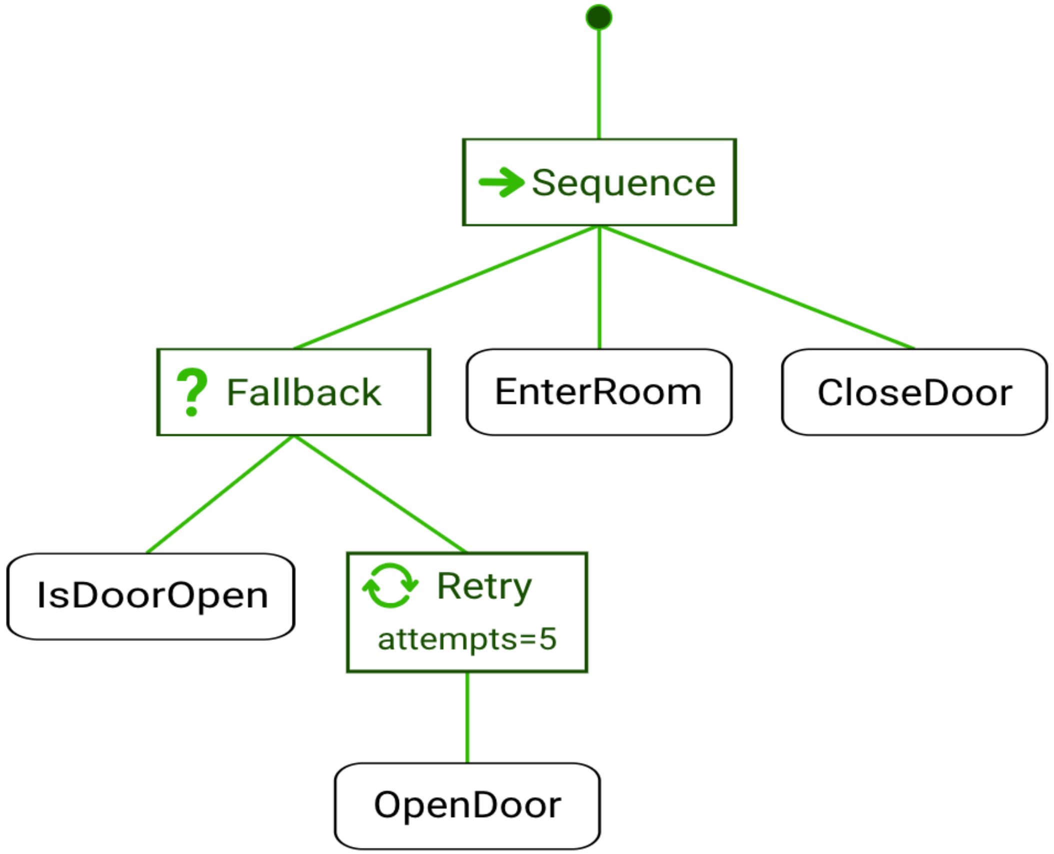

FlexBE App
FlexBE App
Onboard Status:
online
Behavior Dashboard
Statemachine Editor
Runtime Control
Configuration
Add State
Add Behavior
Add Container
Data Flow Graph
Check Behavior
Save Behavior
Undo
Redo
Reset
Hide Comments
Write Comment
Version: 1.0.8
Turtlebot Flex Planner (root)
Receive Goal
GetPoseState
Continue
OperatorDecisionState
New Plan
GetPathState
Operator Action
OperatorDecisionState
Log Success
LogState
Log Recovered
LogState
Receive Path
GetPathState
Container
Concurrency
2 states
Log Fail
LogState
Simple Recovery
Simple Recovery
SimpleRecoverySM
empty
yes
no
finished
failed
done
planned
danger
Execute
DWA
FollowPathState
Health
TurtlebotStatusState
preempted
Simple Recovery
Stop
MoveDistanceState
Backup
MoveDistanceState
Clear Costmap
ClearCostmapsState

BEHAVIOR TREE

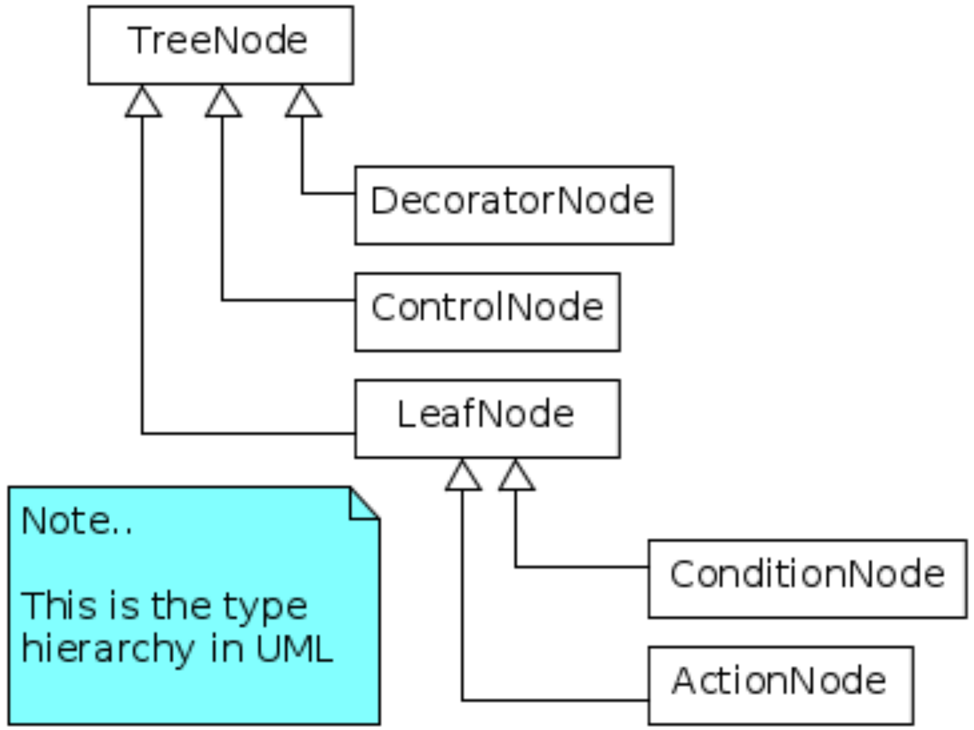
Behavior Tree (BT)

Behavior Trees in Robotics and AI: An Introduction
Michele Colledanchise, Petter Ögren
https://arxiv.org/abs/1709.00084

- Alternative to Finite State Machine (FSM)
  - BT (supposedly) more scalable, more human-understandable and easier to reuse than FSM
  - Intrinsically hierarchical
  - Graphical representation has meaning
  - Expressive
- BehaviorTree CPP V3
  https://www.behaviortree.dev/
- Defined in XML
- Execute top down, left first (similar to DFS)



Types of BT Nodes



Control Node

Decorator Node

Action Node

Condition Node

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"

Example: Navigate To Pose With Replanning and Recovery

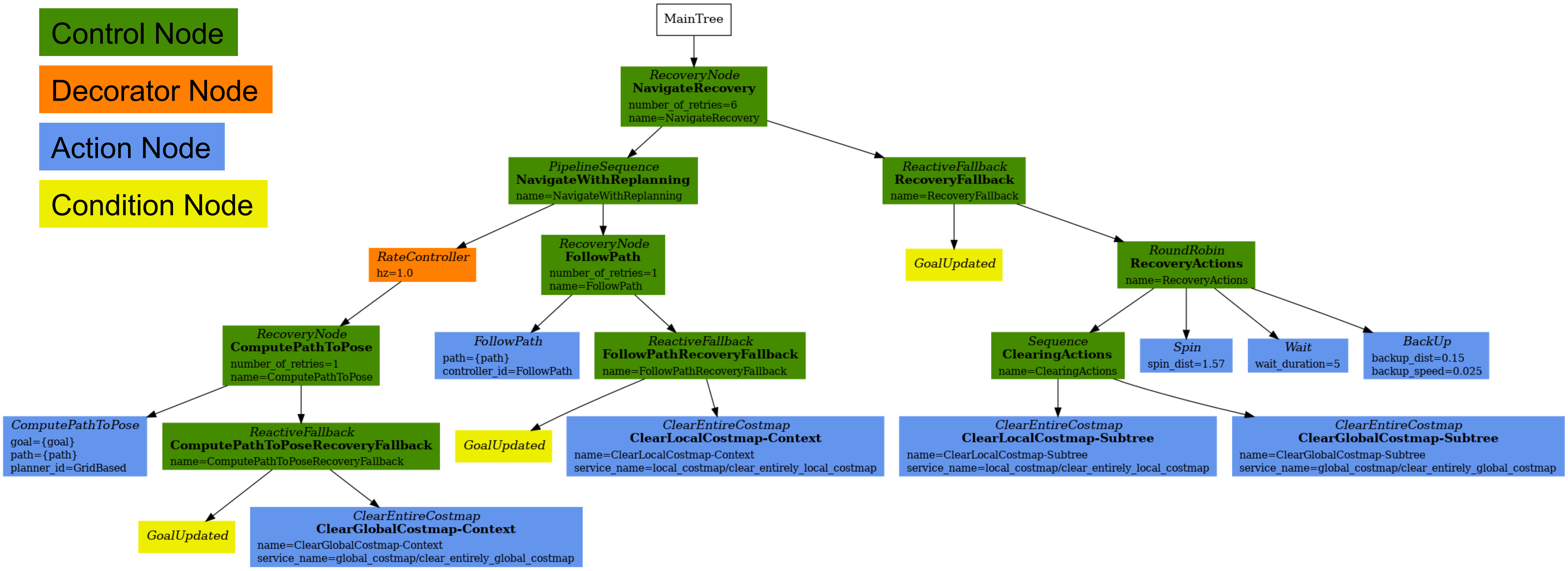
- Tree update rate: 100Hz

🟩 Control Node

🟧 Decorator Node

🟦 Action Node

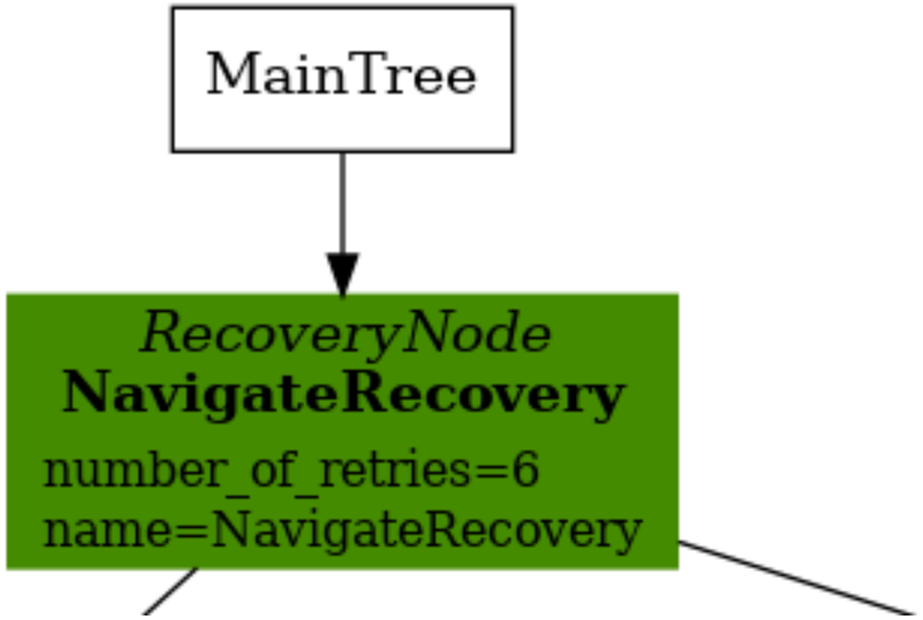
🟨 Condition Node



Recovery Node:

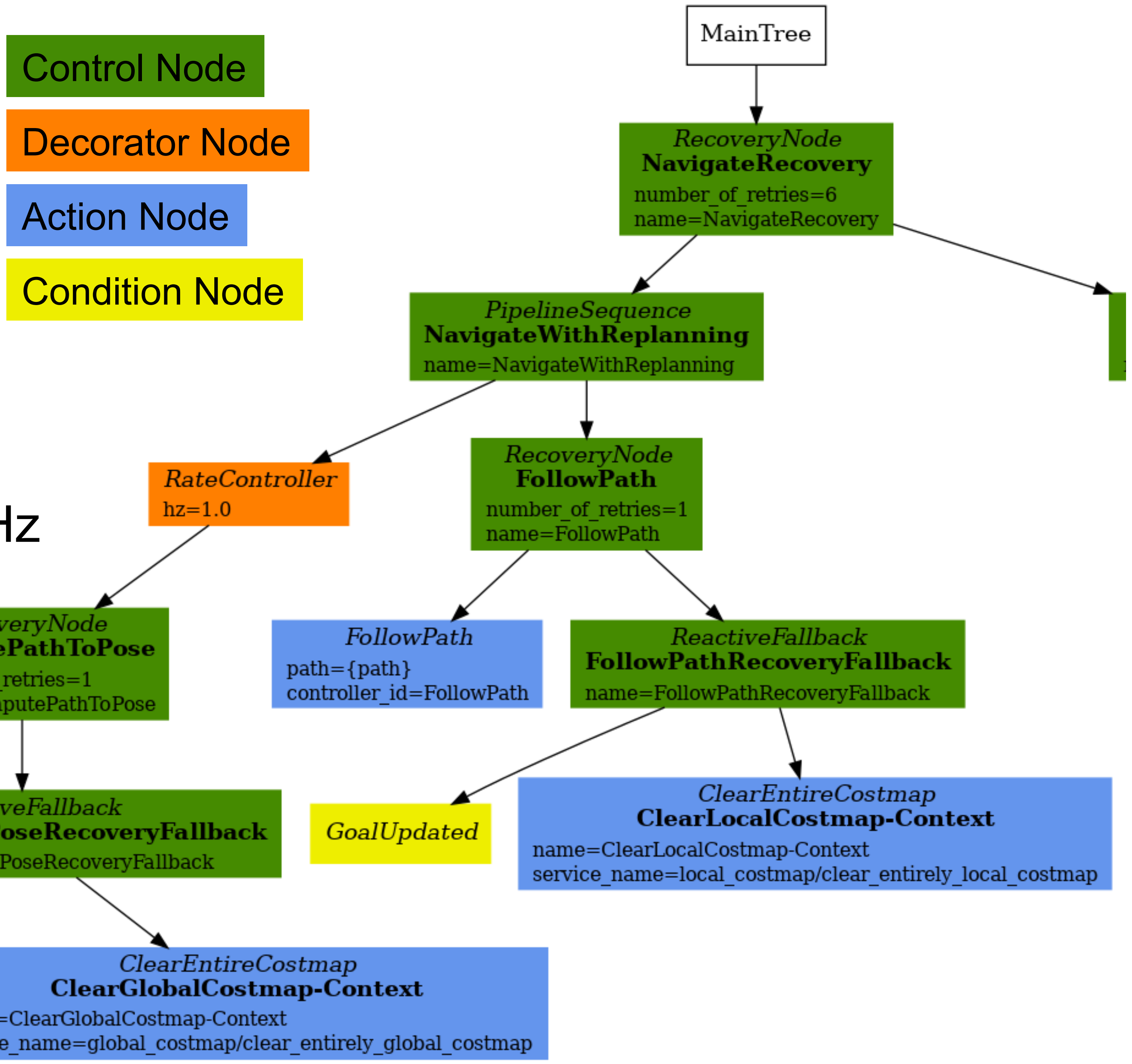
- Node must contain 2 children
- returns success if first succeeds.
- If first fails:
  - Tick the second
  - If successful retry the first
  - Repeat until first returns true or number of retires is up

- Children of Navigate Recovery Node:
  - Navigation Subtree
  - Recovery Subtree

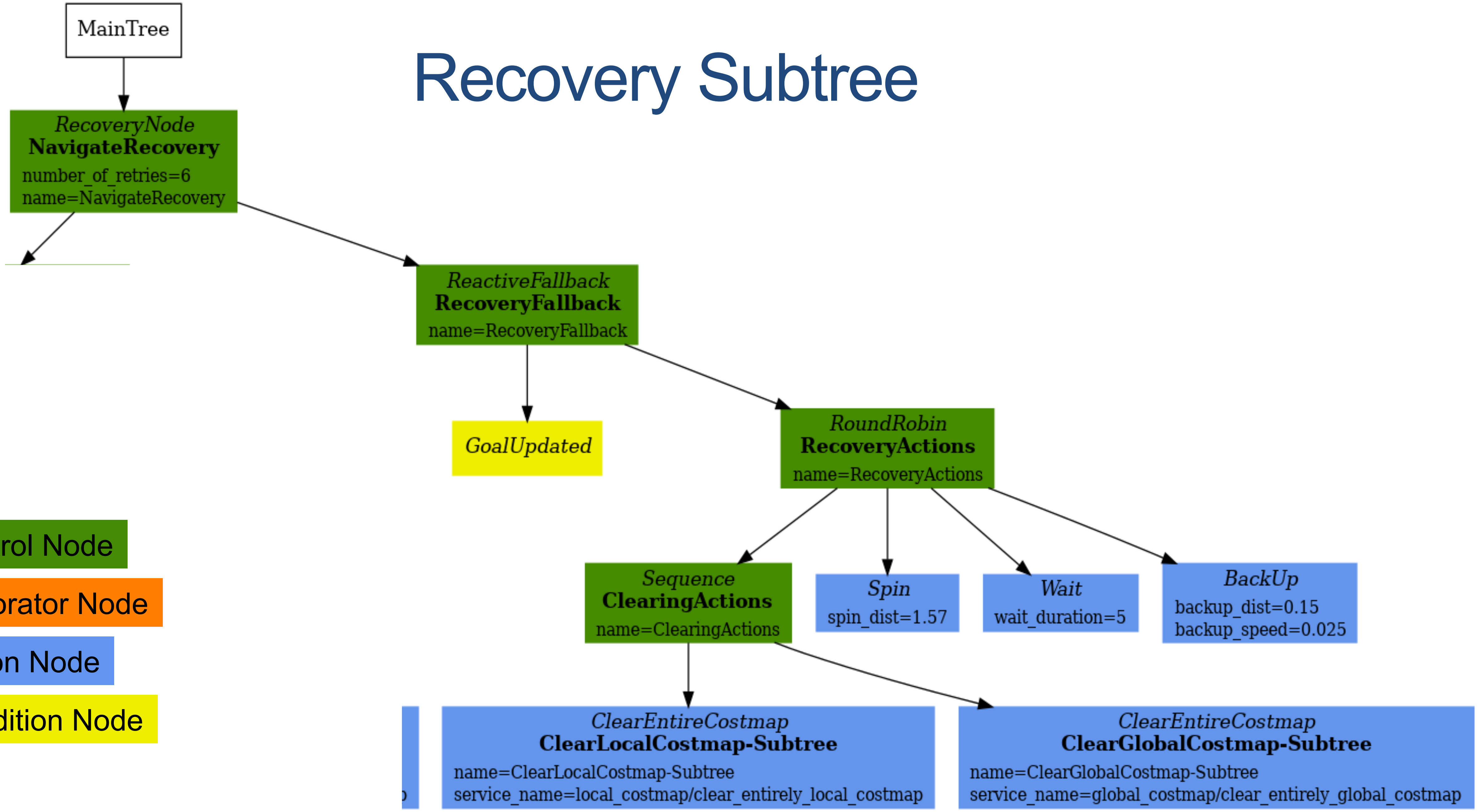


Navigation Subtree

- 2 Subtrees – sequential:
  - Calculate path
  - Follow path
- + recovery behaviors
- RateController: decorate with 1Hz



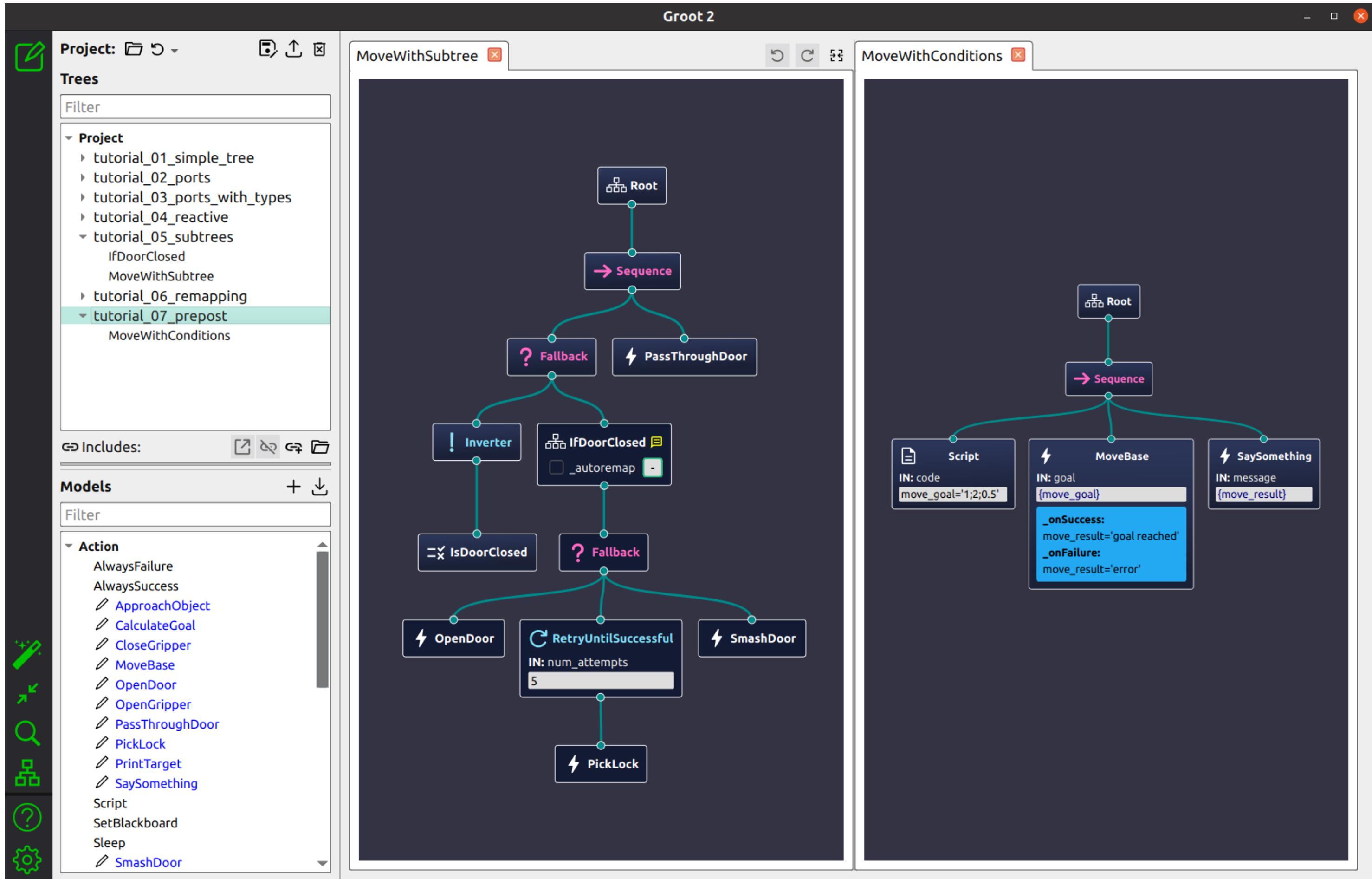
Recovery Subtree



🟩	Control Node
🟧	Decorator Node
🟦	Action Node
🟨	Condition Node

ROS 2 & BT

- Well integrated into ROS 2
  - https://www.behaviortree.dev/docs/ros2_integration/
  - Good support for ROS Actions and ROS Services!
- Groot2: GUI editor for BT
  - https://www.behaviortree.dev/groot



MISSION **PLANNING**

Overview

- Planning
  - PDDL: Language to define the state, problem, …
  - Planner: e.g. STRIPS
    - Precondition and effects on the state space
- Feedback via online replanning
- Sometimes need Task and Motion Planning together:
  - Motion planner would fail if not did a specific action (e.g. remove obstacle) first

Task Planning

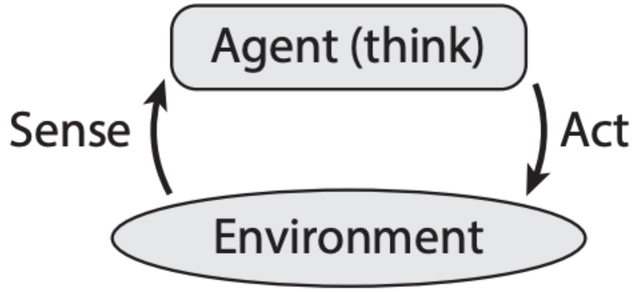
- Task planning – aka Mission Planning – aka automated planning
- 3 approaches:
  - Programming
    - Programmer must anticipate all possible scenarios/ states
  - Learning
    - RL or Imitation learning
    - No guarantees
    - Short horizon/ no complex tasks
  - Model-based approaches
    - Model of the world, reason about it

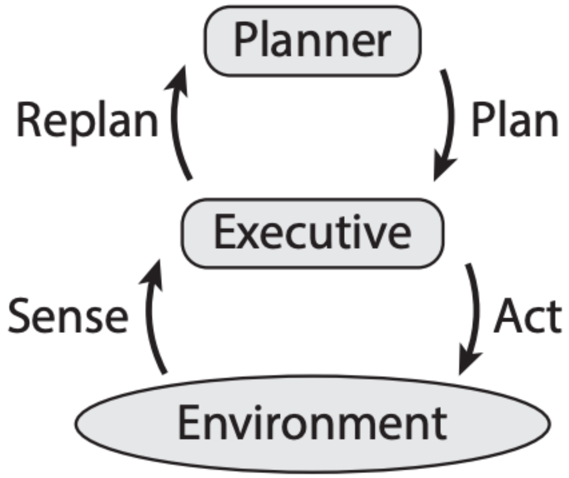
Automated Planning for Robotics
Erez Karpas and Daniele Magazzeni
https://www.annualreviews.org/content/journals/10.1146/annurev-control-082619-100135

Sense/ Think/ Act vs. Execute & Planner

- Assumes:
  - World is deterministic
  - State is fully observable
  - Robot is sole agent
  - Actions are instantaneous

- Planning may take a long time
  - => Multi-level planning (right)

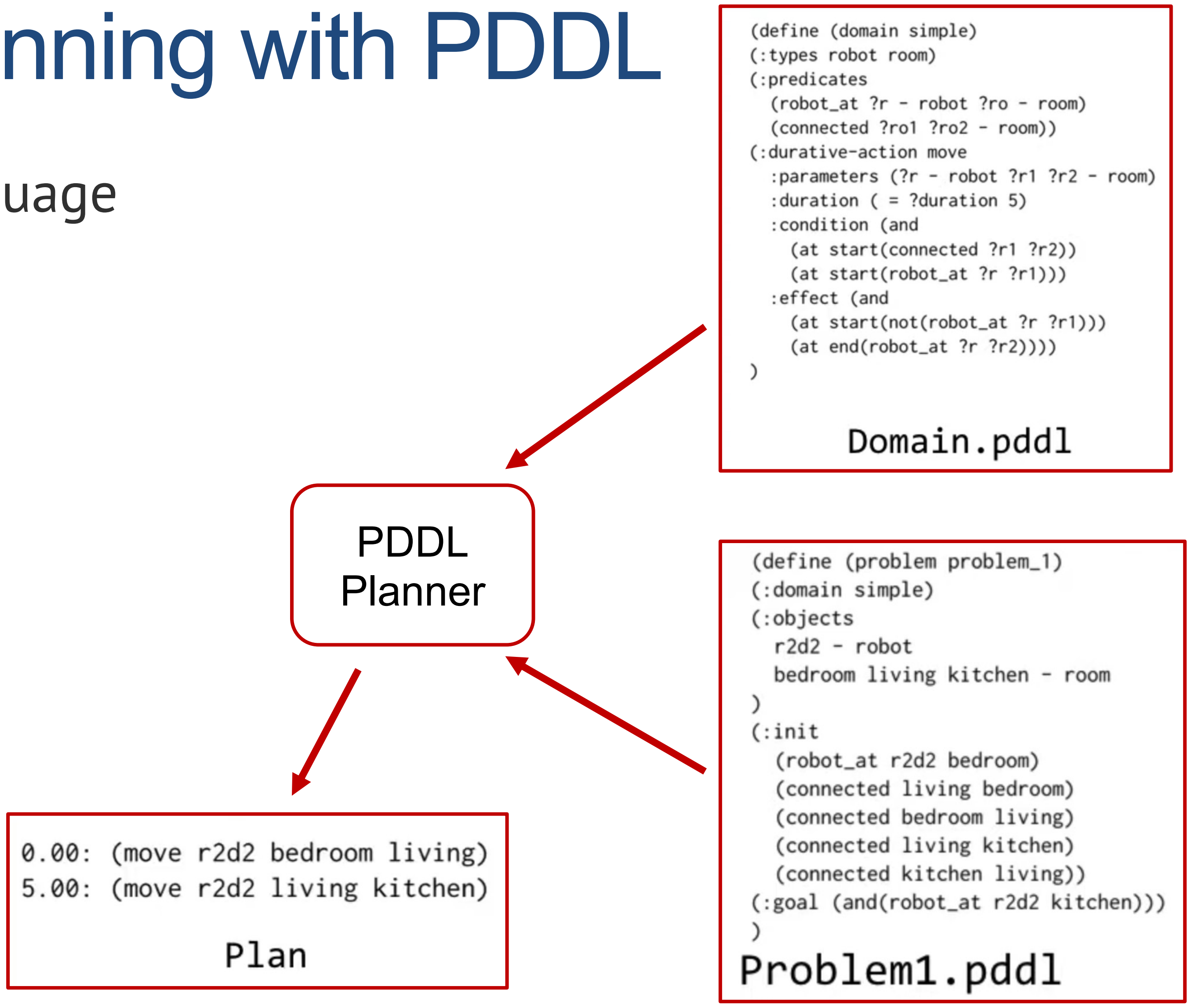




PDDL

Mission and Task Planning with PDDL

- Planning Domain Definition Language
- Establish the rules (Domain)
- Present a situation and goal (Problem)
- Use a plan solver
- Get a plan



```
(define (domain simple)
(:types robot room)
(:predicates
  (robot_at ?r - robot ?ro - room)
  (connected ?ro1 ?ro2 - room))
(:durative-action move
  :parameters (?r - robot ?r1 ?r2 - room)
  :duration ( = ?duration 5)
  :condition (and
    (at start(connected ?r1 ?r2))
    (at start(robot_at ?r ?r1)))
  :effect (and
    (at start(not(robot_at ?r ?r1)))
    (at end(robot_at ?r ?r2))))
)
```

Domain.pddl

```
(define (problem problem_1)
(:domain simple)
(:objects
  r2d2 - robot
  bedroom living kitchen - room
)
(:init
  (robot_at r2d2 bedroom)
  (connected living bedroom)
  (connected bedroom living)
  (connected living kitchen)
  (connected kitchen living))
(:goal (and(robot_at r2d2 kitchen)))
)
```

Problem1.pddl

```
0.00: (move r2d2 bedroom living)
5.00: (move r2d2 living kitchen)
```

Plan

What is PDDL?

PDDL = Planning Domain Definition Language

- standard encoding language for “classical” planning tasks

Components of a PDDL planning task:

- **Objects:** Things in the world that interest us.
- **Predicates:** Properties of objects that we are interested in; can be *true* or *false*.
- **Initial state:** The state of the world that we start in.
- **Goal specification:** Things that we want to be true.
- **Actions/Operators:** Ways of changing the state of the world.

Malte Helmert https://www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf

How to Put the Pieces Together

Planning tasks specified in PDDL are separated into two files:

1. A **domain file** for predicates and actions.
2. A **problem file** for objects, initial state and goal specification.

Domain Files

Domain files look like this:

```
(define (domain <domain name>)
      <PDDL code for predicates>
      <PDDL code for first action>
      [...]
      <PDDL code for last action>
)
```

`<domain name>` is a string that identifies the planning domain, e.g. `gripper`.

Example on the web: `gripper.pddl`.

Problem Files

Problem files look like this:

```
(define (problem <problem name>)
   (:domain <domain name>)
       <PDDL code for objects>
       <PDDL code for initial state>
       <PDDL code for goal specification>
)
```

`<problem name>` is a string that identifies the planning task, e.g. `gripper-four-balls`.

`<domain name>` must match the domain name in the corresponding domain file.

Running Example

Gripper task with four balls:

There is a robot that can move between two rooms and pick up or drop balls with either of his two arms. Initially, all balls and the robot are in the first room. We want the balls to be in the second room.

- **Objects:** The two rooms, four balls and two robot arms.
- **Predicates:** Is x a room? Is x a ball? Is ball x inside room y? Is robot arm x empty? [...]
- **Initial state:** All balls and the robot are in the first room. All robot arms are empty. [...]
- **Goal specification:** All balls must be in the second room.
- **Actions/Operators:** The robot can move between rooms, pick up a ball or drop a ball.

Gripper task: Objects

Objects:

Rooms: `rooma`, `roomb`

Balls: `ball1`, `ball2`, `ball3`, `ball4`

Robot arms: `left`, `right`

In PDDL:

```
(:objects rooma roomb
       ball1 ball2 ball3 ball4 left
       right)
```

Gripper task: Predicates

Predicates:

ROOM(x)	–true iff x is a room
BALL(x)	–true iff x is a ball
GRIPPER(x)	–true iff x is a gripper (robot arm)
at-robby(x)	–true iff x is a room and the robot is in x
at-ball(x, y)	–true iff x is a ball, y is a room, and x is in y
free(x)	–true iff x is a gripper and x does not hold a ball
carry(x, y)	–true iff x is a gripper, y is a ball, and x holds y

In PDDL:

```
(:predicates (ROOM ?x) (BALL ?x)
             (GRIPPER ?x) (at-robby ?x)
             (at-ball ?x ?y)
             (free ?x) (carry ?x ?y))
```

Gripper task: Initial state

- Initial state:
- `ROOM(rooma)` and `ROOM(roomb)` are true.
- `BALL(ball1)`, ..., `BALL(ball4)` are true.
- `GRIPPER(left)`, `GRIPPER(right)`, `free(left)` and `free(right)` are true.
- `at-robby(rooma)`, `at-ball(ball1, rooma)`, ..., `at-ball(ball4, rooma)` are true. Everything else is false.

- In PDDL:

```
(:init (ROOM rooma) (ROOM roomb)
      (BALL ball1) (BALL ball2) (BALL ball3) (BALL ball4)
      (GRIPPER left) (GRIPPER right)    (free left) (free right)
      (at-robby rooma)
      (at-ball ball1 rooma) (at-ball    ball2 rooma)
      (at-ball ball3 rooma) (at-ball    ball4 rooma))
```

Gripper task: Goal specification

Goal specification:

`at-ball(ball1, roomb)`, ..., `at-ball(ball4, roomb)` must be true. Everything else we don't care about.

In PDDL:

```
    (:goal (and (at-ball ball1 roomb)
                (at-ball ball2 roomb)
                (at-ball baxll3 roomb)
                (at-ball ball4 roomb)))
```

Gripper task: Movement operator

Action/Operator:

Description: The robot can move from x to y.

Precondition: `ROOM`(x), `ROOM`(y) and `at-robby`(x) are true.

Effect: `at-robby`(y) becomes true. `at-robby`(x) becomes false. Everything else doesn't change.

In PDDL:

```
(:action move :parameters (?x ?y)
   :precondition (and (ROOM ?x) (ROOM ?y)
                      (at-robby ?x))
    :effect           (and (at-robby ?y)
                           (not (at-robby
                           ?x))))
```

Gripper task: Pick-up operator

Action/Operator:

Description: The robot can pick up x in y with z.
Precondition: BALL(x), ROOM(y), GRIPPER(z), at-ball(x, y), at-robby(y) and free(z) are true.
Effect: carry(z, x) becomes true. at-ball(x, y) and free(z) become false. Everything else doesn't change.

In PDDL:

```
(:action pick-up :parameters (?x ?y ?z)
   :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                      (at-ball ?x ?y) (at-robby ?y) (free ?z))
   :effect       (and (carry ?z ?x)
                      (not (at-ball ?x ?y)) (not (free ?z))))
```

Gripper task: Drop operator

Action/Operator:

Description: The robot can drop x in y from z.

(Preconditions and effects similar to the pick-up operator.)

In PDDL:

```
(:action drop :parameters (?x ?y ?z)
   :precondition (and (BALL ?x) (ROOM ?y) (GRIPPER ?z)
                      (carry ?z ?x) (at-robby ?y))
    :effect           (and (at-ball ?x ?y) (free ?z)
                           (not (carry ?z ?x))))
```

A Note on Action Effects

Action effects can be more complicated than seen so far.

They can be **universally quantified**:

```
(forall (?v1 ... ?vn)
        <effect>)
```

They can be **conditional**:

```
(when <condition>
      <effect>)
```

Solver/ Planners

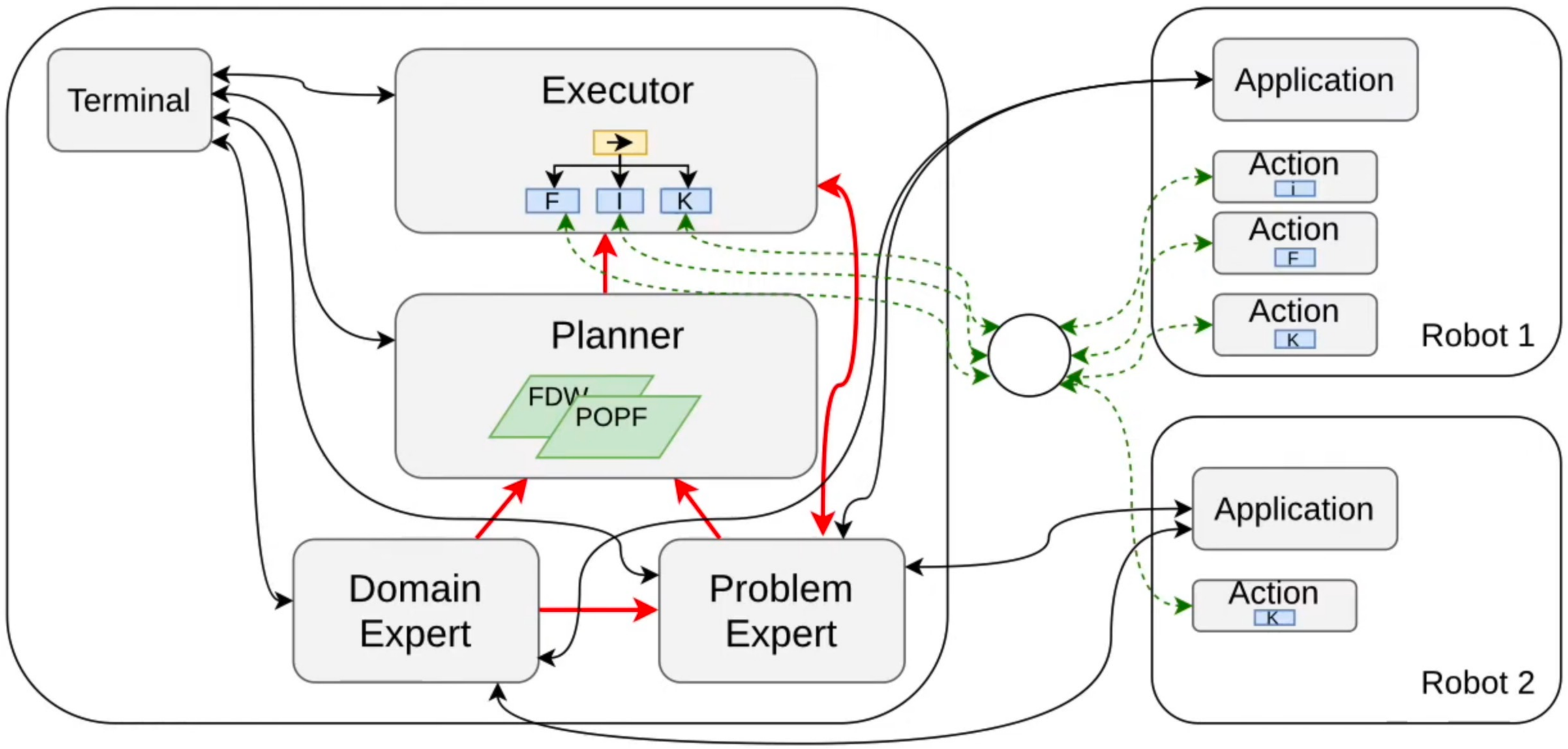
- PDDL just a description language
- Standardized - to be used in different solvers
- Many different algorithms:
  - Classical planning:
    - forward chaining of state space, maybe with heuristics
    - Backward chaining (e.g. STRIPS from 1970s)
  - Reduction to other problems
    - Model checking (planning is a subclass of model checking)
  - Temporal planning
    - Actions may be executed in parallel/ overlapping in time
  - Probabilistic planning
    - MDP
    - POMDP

ROS 2 Planning System

PlanSys2

- Framework for AI Planning
- Modular Design
- PlanSyst2 Terminal
- Multirobot
- Parallel execution
- Load balancing

- Planners as plugins
- Default:
  - Partial Order Planning Forwards (POPF)
  - forwards-chaining temporal planner



https://plansys2.github.io/