

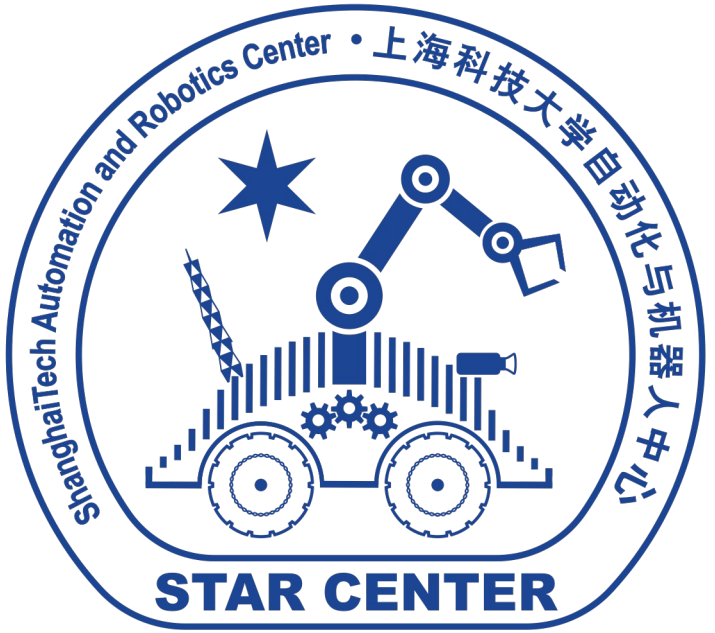
上海科技大学

ShanghaiTech University

CS289: **Mobile Manipulation Fall 2026**

Sören Schwertfeger

ShanghaiTech University



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

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

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:

inputs −→ outputs

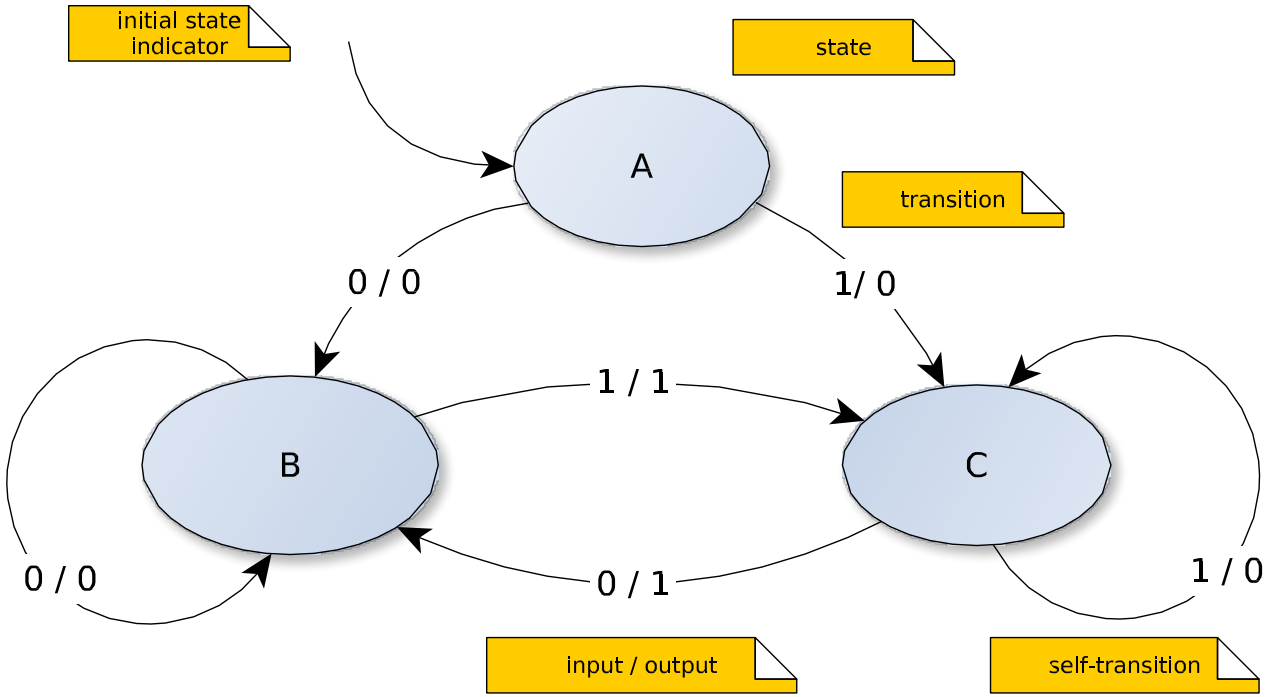
0 1 1 1 −→ 0 1 0 0

0 1 1 1 1 0 −→ 0 1 0 0 0 1

1 0 1 0 1 0 −→ 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

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

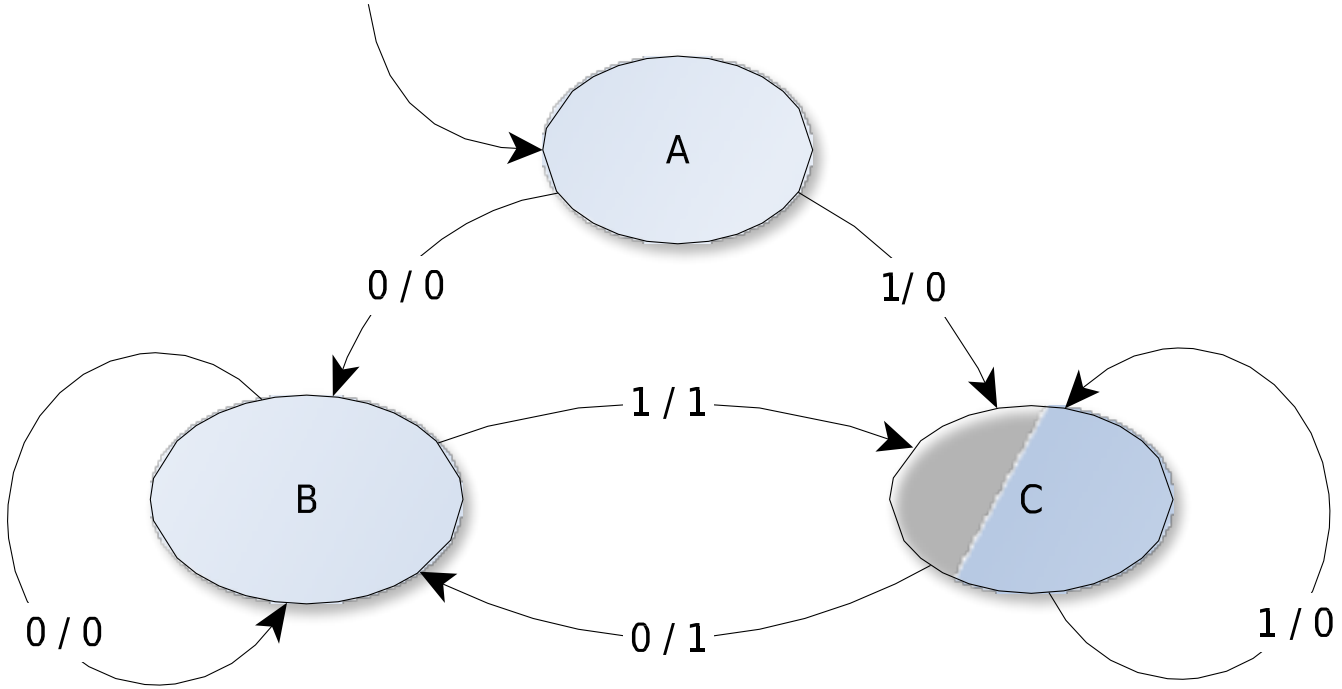
where:

- Σ (Sigma) is the input alphabet (a finite, non-empty set of symbols).
- Γ (Gamma) 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 (Delta) 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.

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.

Mealy vs. Moore FSMs

- **Mealy:** Output depends on the current state and input; write it on the transition (input / output). It can respond in the same reaction and often uses fewer states.
- **Moore:** Output depends only on the current state; write it inside the state. The output is set before the transition, so the machine is strictly causal.
- A Moore machine can be converted to an equivalent Mealy machine. Converting Mealy to Moore may delay the output by one reaction.

Determinacy

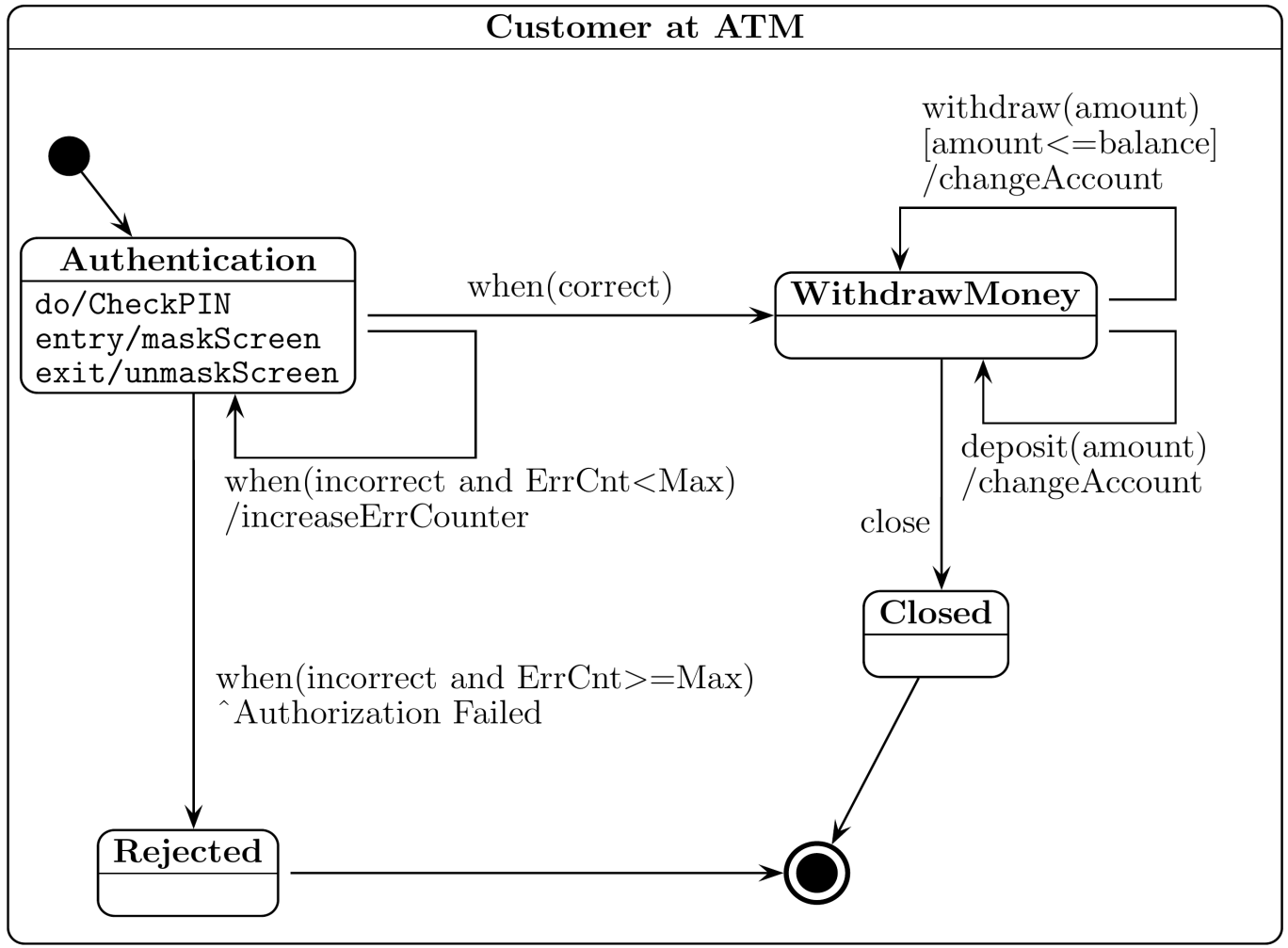
- A state machine is said to be **deterministic** (or **determinate**) if, for each state, there is at most one transition enabled by each input value.
- The given formal definition of an FSM ensures that it is deterministic, since the transition function δ is a function, not a one-to-many mapping.
- 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*.

Implementation: imperative programming language

```
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;
}
```

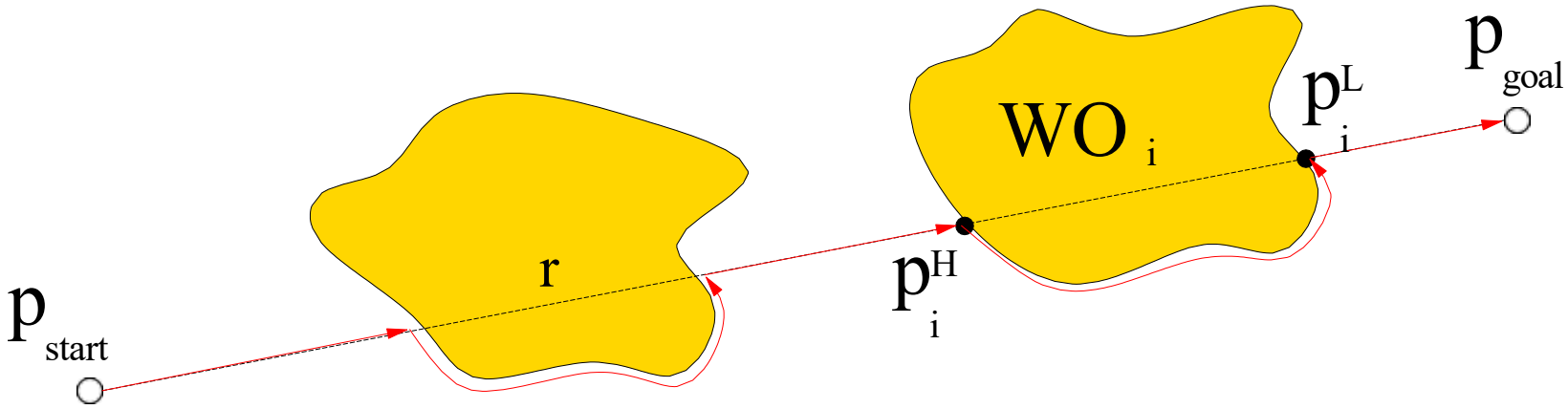
Implementation: UML State Machine Diagram

Example: ATM



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

Example: Bug - 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

Example: Bug - 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 - hypoteses (2/2)

- State variables:

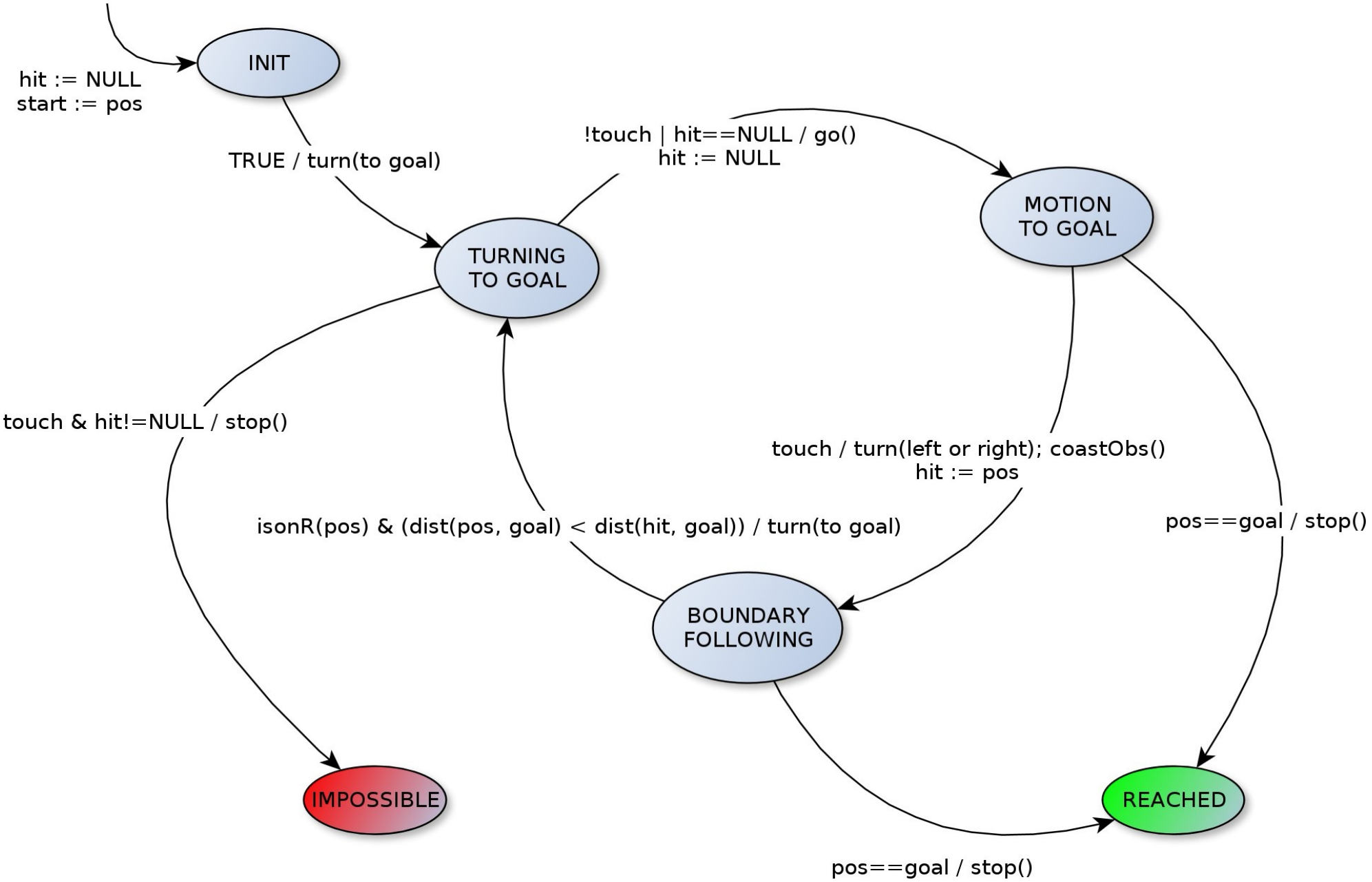
  hit - variable in $W \cup \{NULL\}$, which stores the hit point

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

- Parameter:

  goal - constant in W

Bug
Mealy FSM

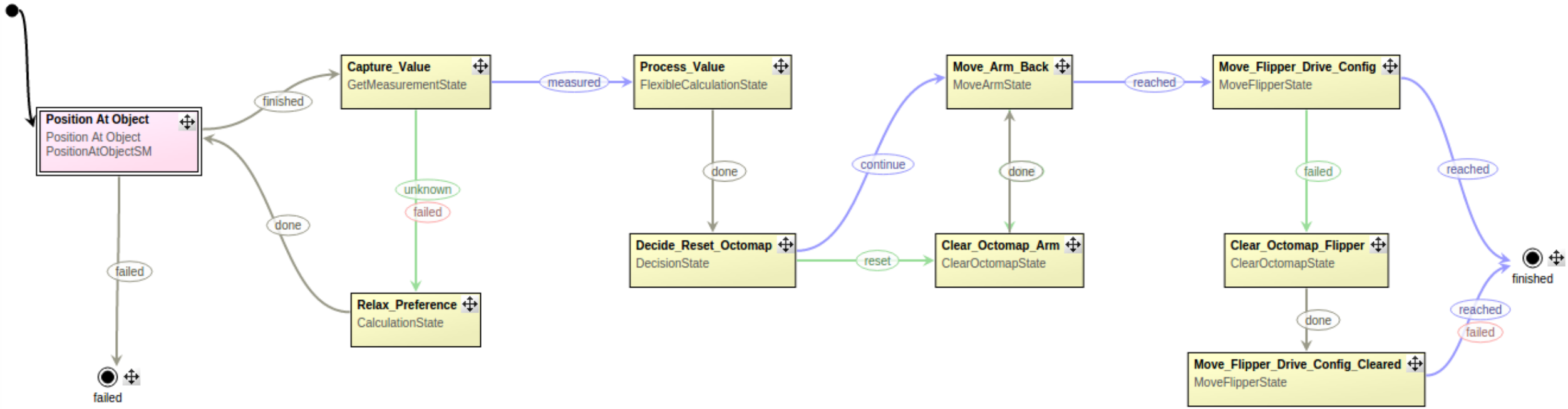


ROS: FLEXBE

FlexBe

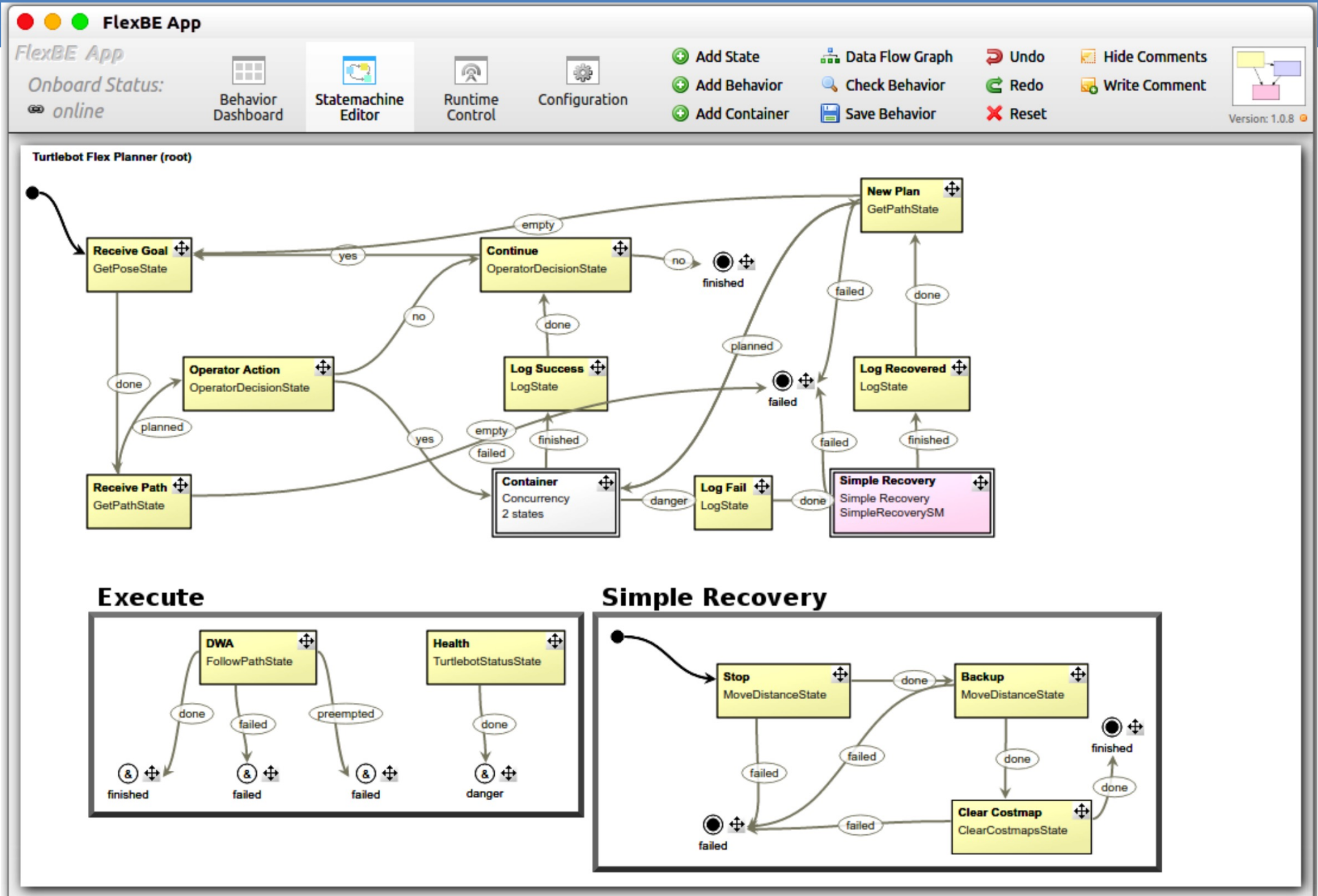
- FlexBE is a powerful and user-friendly high-level behavior engine, flexibly applicable to numerous systems and scenarios.
- drag&drop behavior creation
- automated code generation
- monitor behavior execution
- adjustable level of autonomy
- runtime-modification of behaviors

- 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/

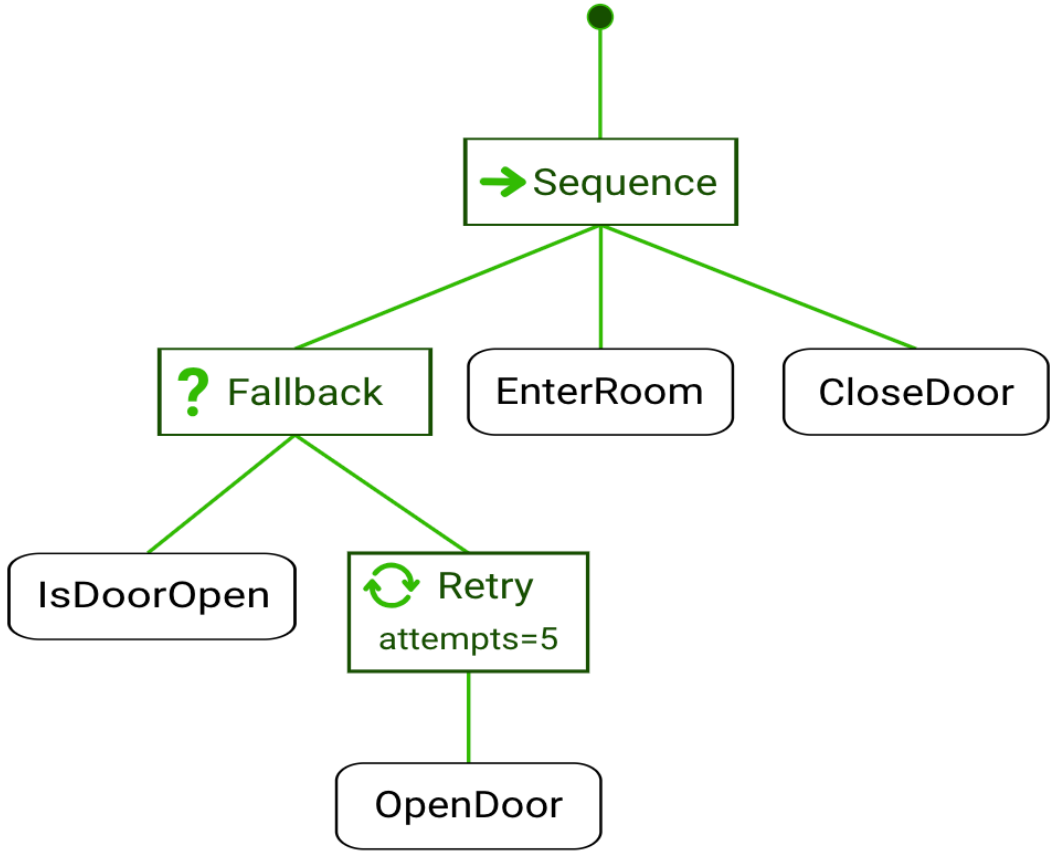

FlexBE App
FlexBE App
Onboard Status:
online
Behavior Dashboard
Statemachine Editor
Runtime Control
Configuration
Add State
Add Behavior
Add Container
Data Flow Graph
Check Behavior
Save Behavior
Undo
Redo
Reset
Hide Comments
Write Comment
Version: 1.0.8
Turtlebot Flex Planner (root)
Receive Goal
GetPoseState
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
finished
failed
empty
yes
no
done
planned
danger
Execute
DWA
FollowPathState
Health
TurtlebotStatusState
preempted
finished
failed
failed
danger
Simple Recovery
Stop
MoveDistanceState
Backup
MoveDistanceState
Clear Costmap
ClearCostmapsState
finished
failed

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)



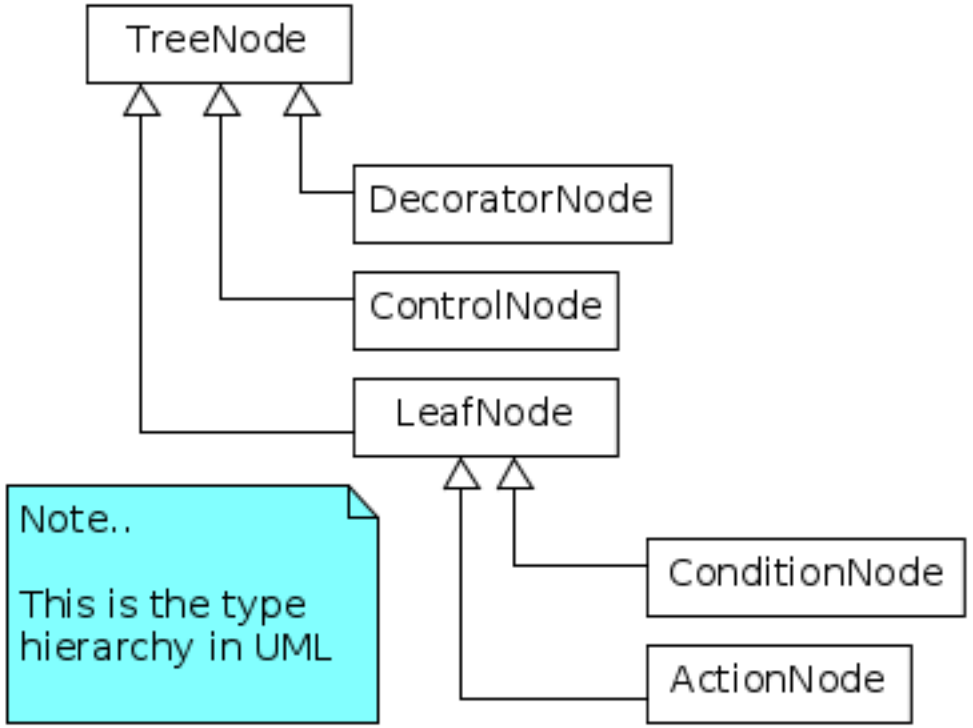
Types of BT Nodes

Control Node

Decorator Node

Action Node

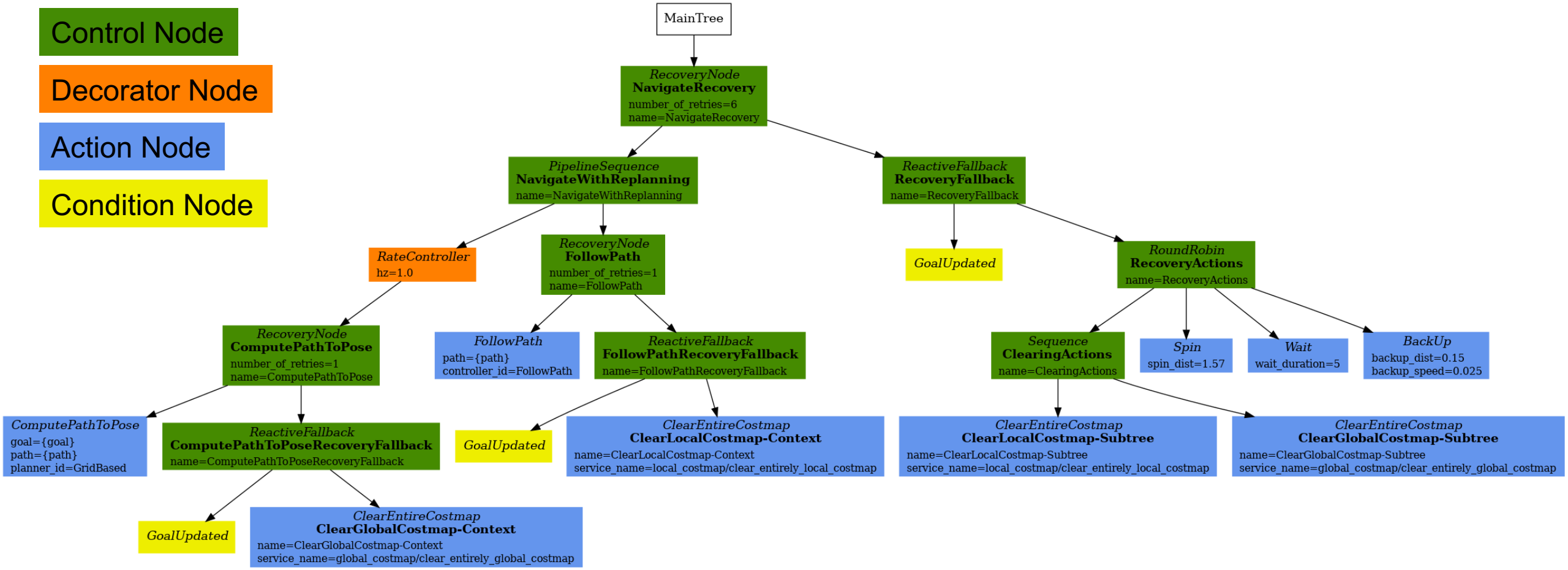
Condition Node



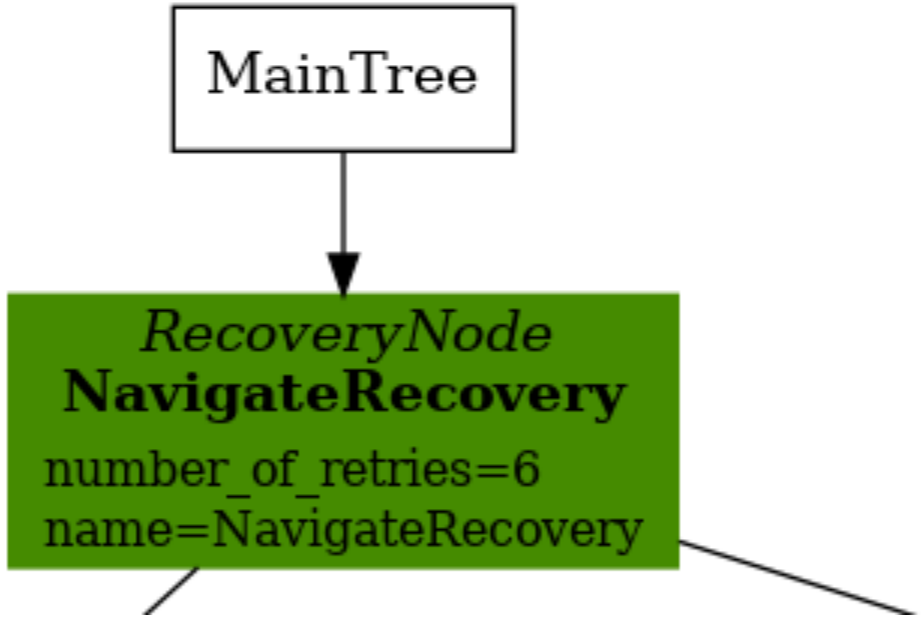
Type of TreeNode	Children Count	Notes
ControlNode	1...N	Usually, ticks a child based on the result of its siblings or/and its own state.
DecoratorNode	1	Among other things, it may alter the result of its child or tick it multiple times.
ConditionNode	0	Should not alter the system. Shall not return RUNNING.
ActionNode	0	This is the Node that "does something"

Example: Navigate To Pose With Replanning and Recovery

- Tree update rate: 100Hz



Recovery Node:

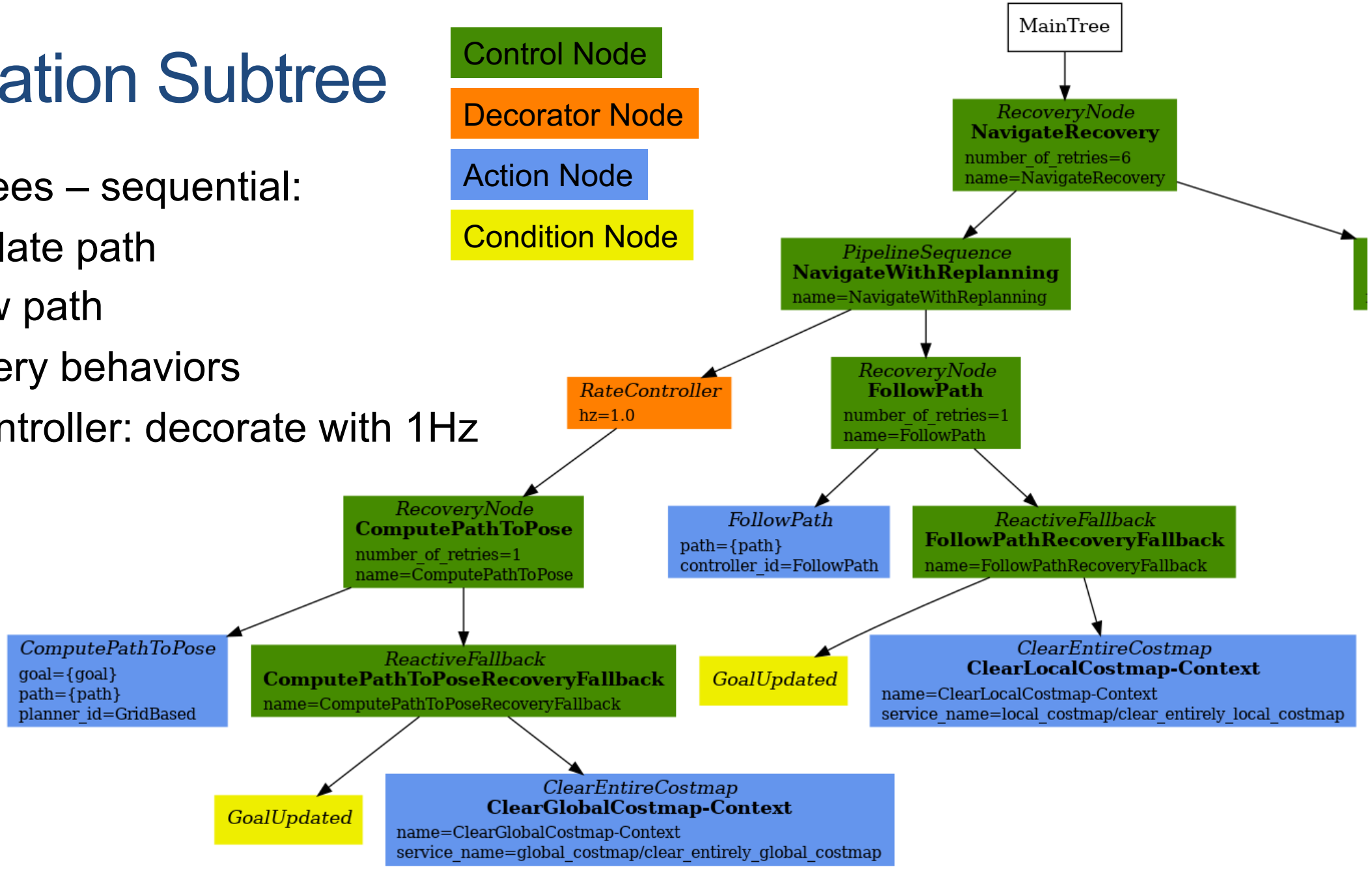


- Node must contain 2 children
- returns success if first succeeds.
- If first fails:
  - Tick the second
  - If successful retry the first
  - Repeat until first returns true or number of retires is up

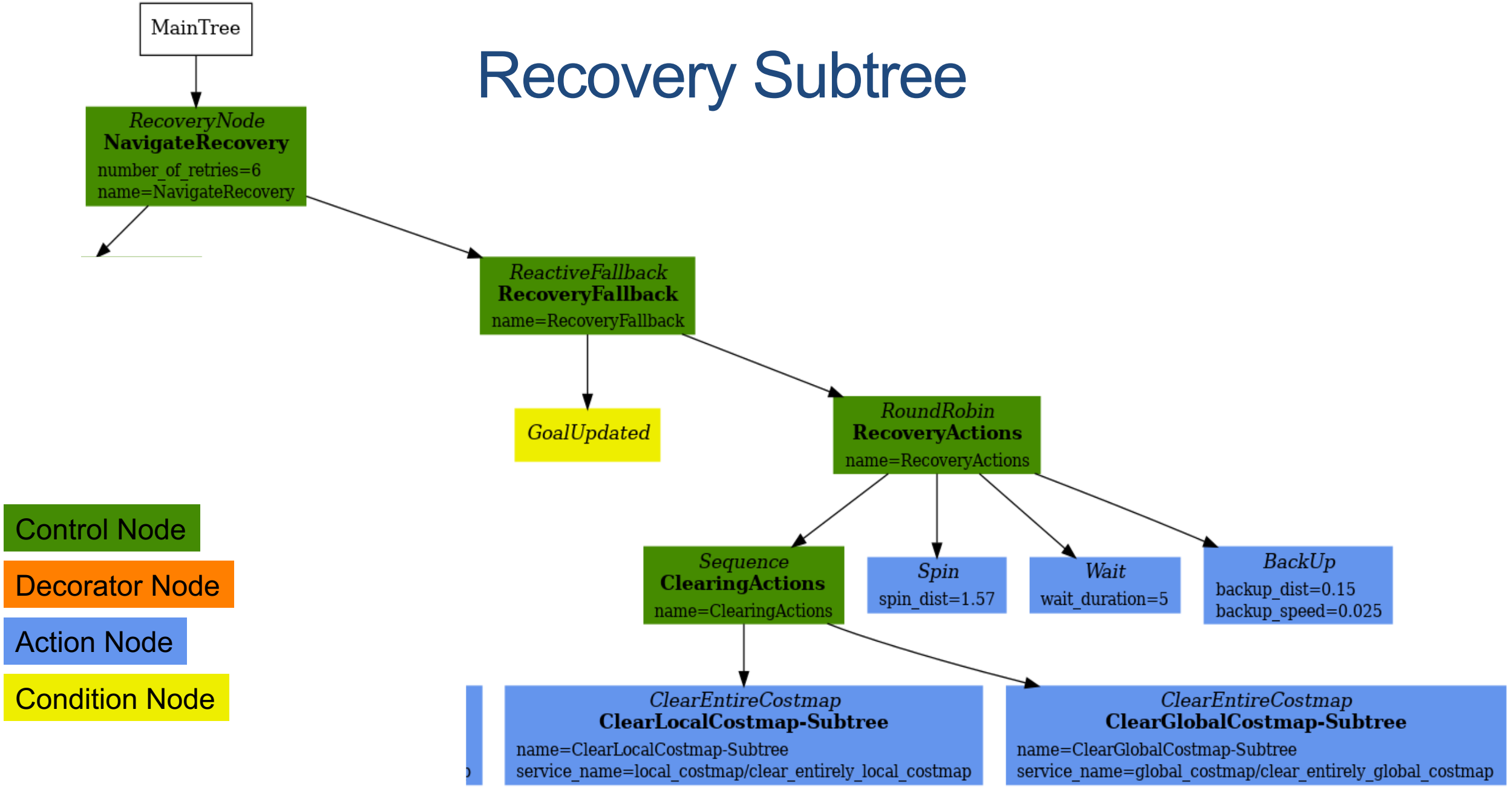
- Children of Navigate Recovery Node:
  - Navigation Subtree
  - Recovery Subtree

Navigation Subtree

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

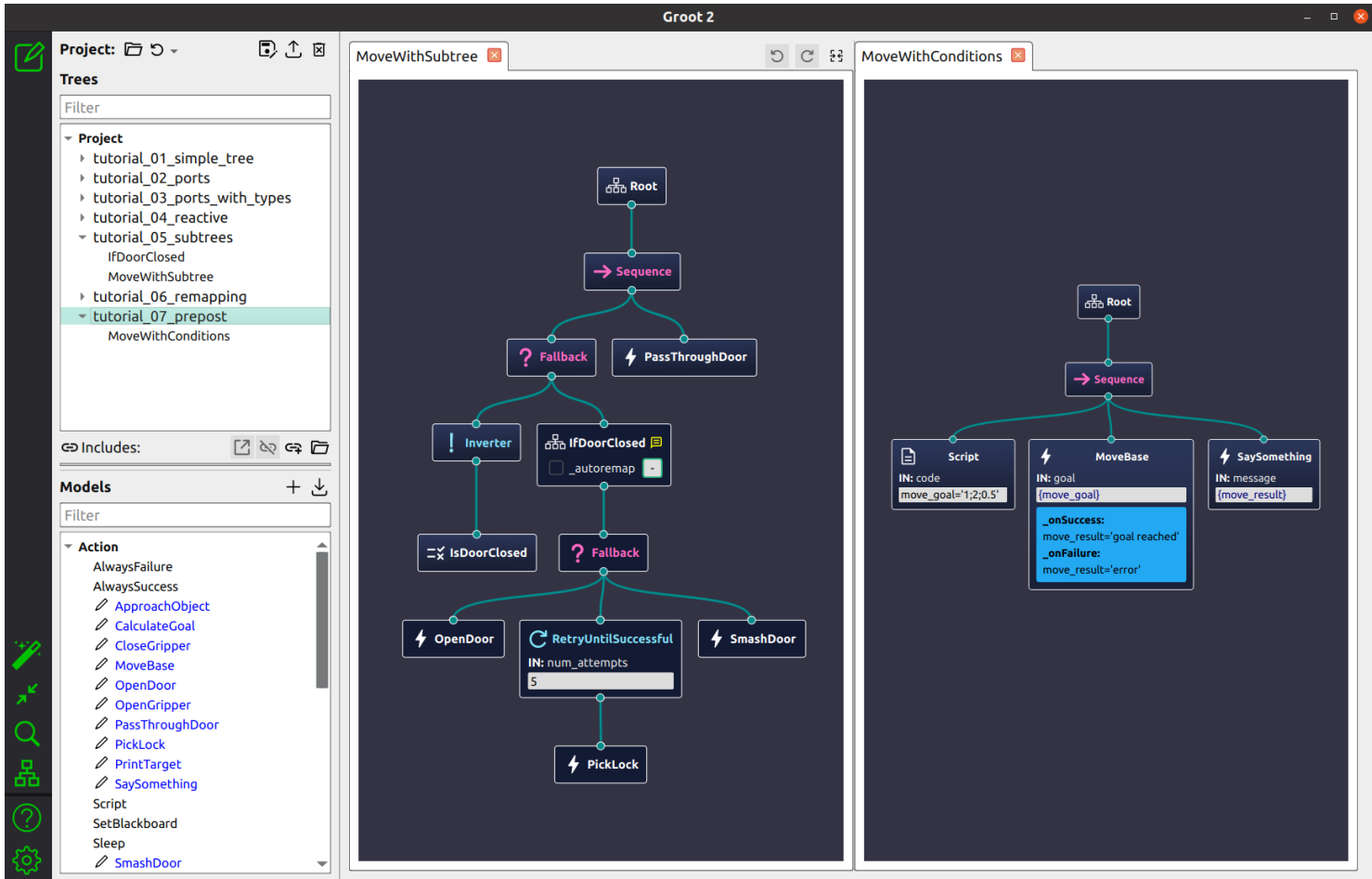


Recovery Subtree



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

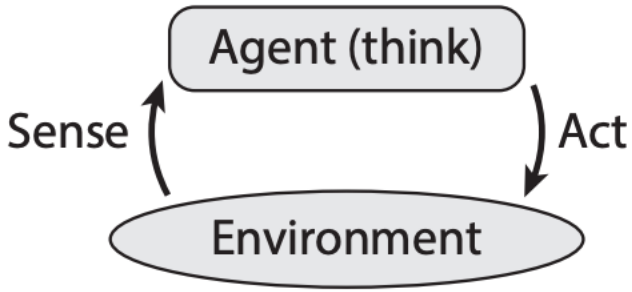
Task Planning

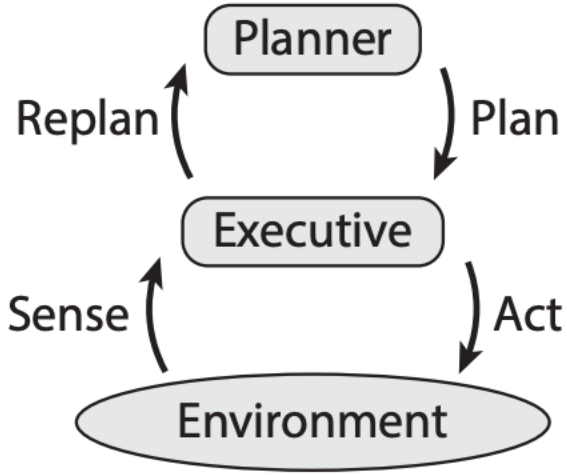
- Task planning – aka Mission Planning – aka automated planning
- 3 approaches:
  - Programming
    - Programmer must anticipate all possible scenarios/ states
  - Learning
    - RL or Imitation learning
    - No guarantees
    - Short horizon/ no complex tasks
  - Model-based approaches
    - Model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

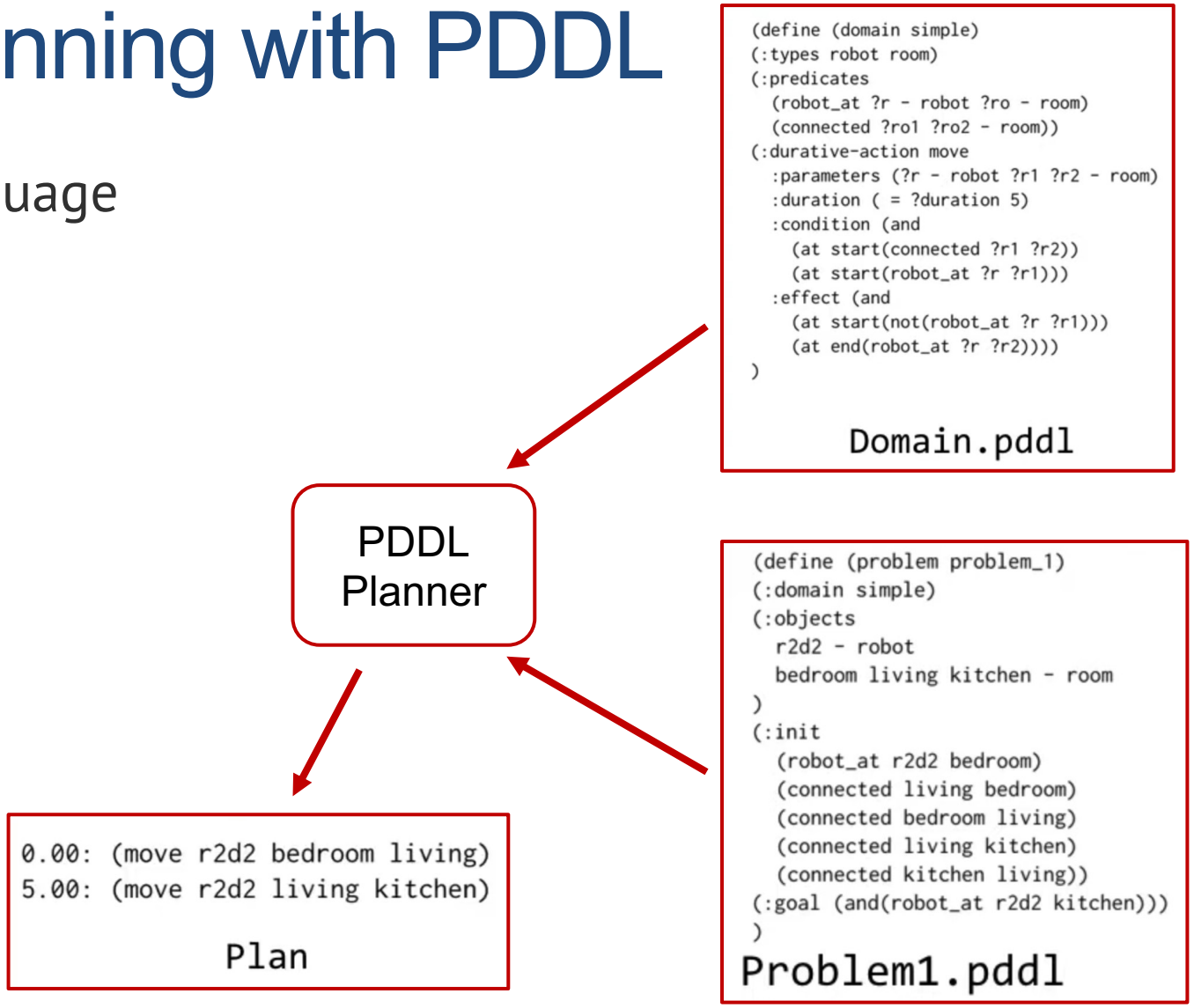




PDDL

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



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 ball3 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>)
```

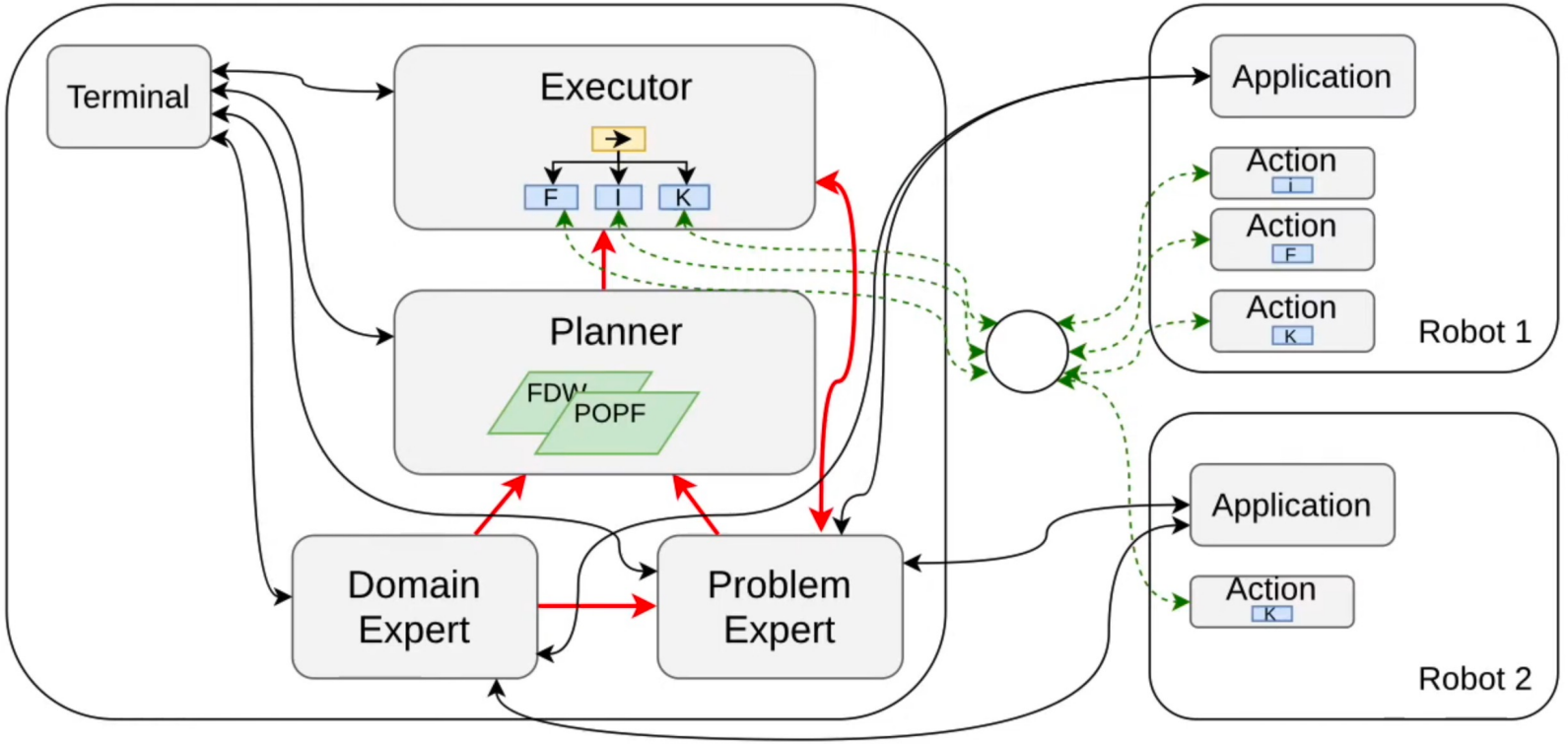
Solver/ Planners

- 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

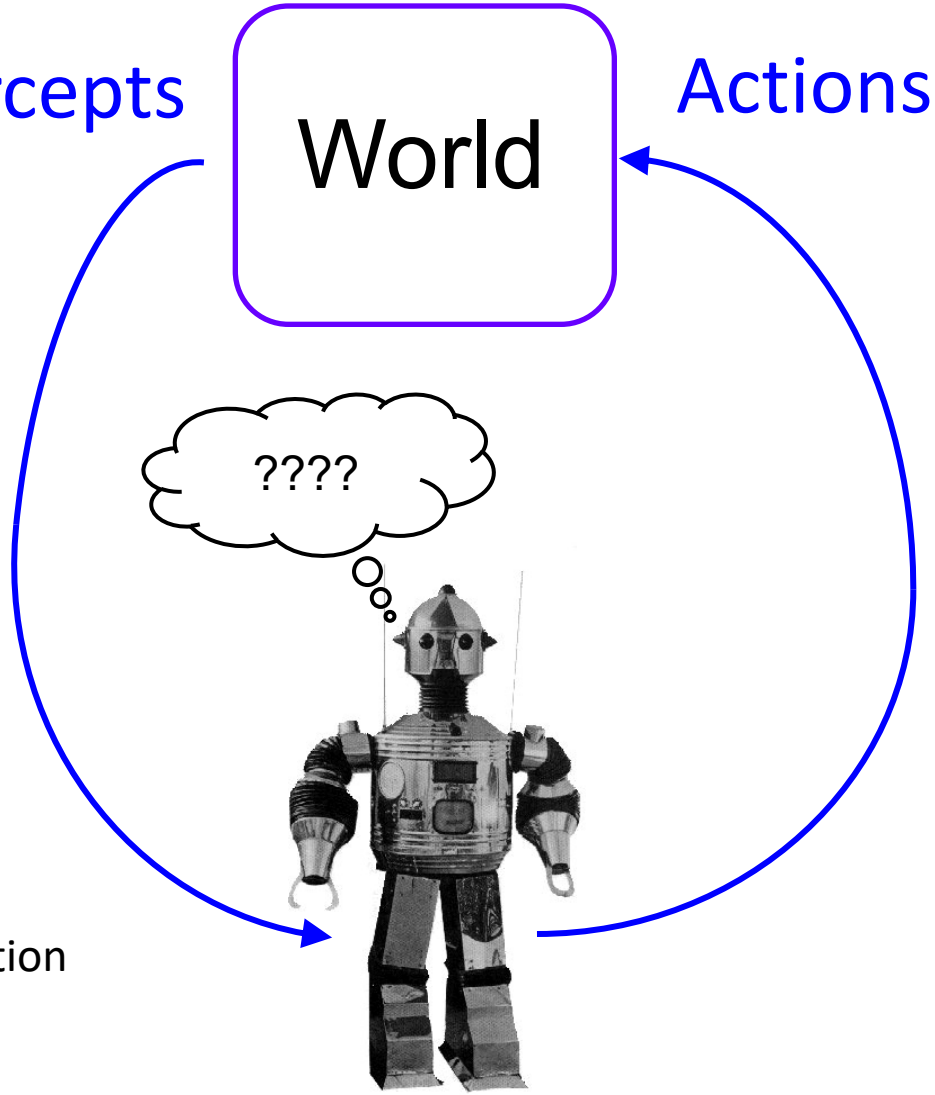
- Framework for AI Planning
- Modular Design
- PlanSyst2 Terminal
- Multirobot
- Parallel execution
- Load balancing



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

https://plansys2.github.io/

Classical Planning Assumptions



perfect

fully observable

sole source of change

deterministic

instantaneous

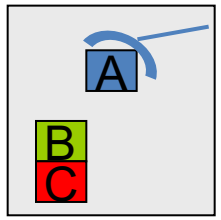
Goal

achieve goal condition

Material: Alan Fern; Daniel Weld
Oregon State University
https://web.engr.oregonstate.edu/~afern/classes/cs533/notes/strips-intro.pptx

Representing States

World states are represented as sets of facts.
We will also refer to facts as propositions.

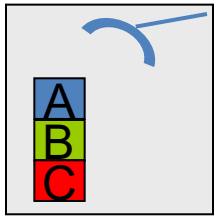


holding(A)
clear(B)
on(B,C)
onTable(C)

State 1

handEmpty
clear(A)
on(A,B)
on(B,C)
onTable(C)

State 2

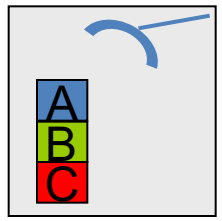


Closed World Assumption (CWA):
Fact not listed in a state are assumed to be false. Under CWA we are assuming the agent has full observability.

Representing Goals

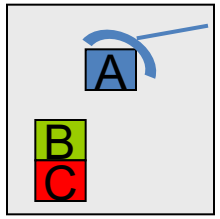
Goals are also represented as sets of facts.
For example { on(A,B) } is a goal in the blocks world.

A **goal state** is any state that contains all the goal facts.



handEmpty
clear(A)
on(A,B)
on(B,C)
onTable(C)

State 1

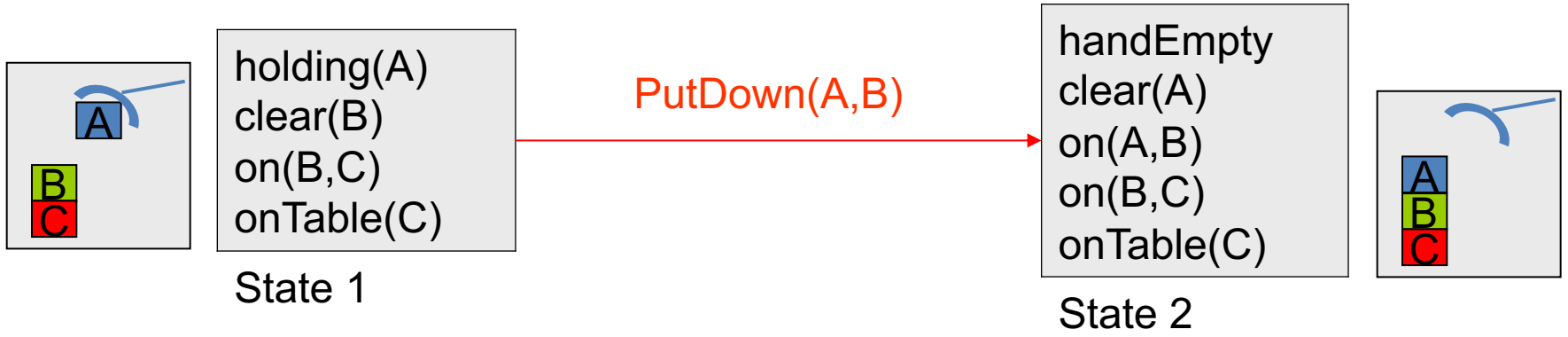


holding(A)
clear(B)
on(B,C)
onTable(C)

State 2

State 1 is a goal state for the goal { on(A,B) }.
State 2 is not a goal state for the goal { on(A,B) }.

Representing Action in STRIPS



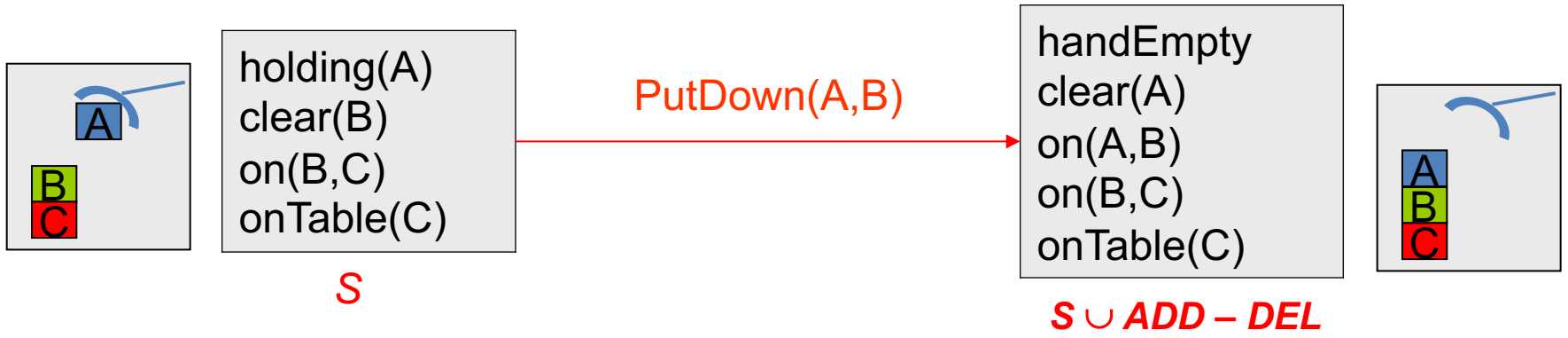
A STRIPS action definition specifies:

1) a set PRE of preconditions facts
2) a set ADD of add effect facts
3) a set DEL of delete effect facts

PutDown(A,B):

PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A) }
DEL: { holding(A), clear(B) }

Semantics of STRIPS Actions



- A STRIPS action is **applicable** (or allowed) in a state when its preconditions are contained in the state.
- Taking an action in a state ***S*** results in a new state ***S ∪ ADD – DEL*** (i.e. add the add effects and remove the delete effects)

PutDown(A,B):

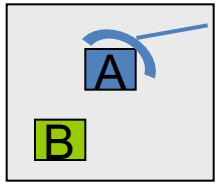
PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A)}
DEL: { holding(A), clear(B) }

STRIPS Planning Problem

A **STRIPS planning problem** specifies:
1) an initial state *S*
2) a goal *G*
3) a set of STRIPS actions

Objective: find a "short" action sequence reaching a goal state, or report that the goal is unachievable

Example Problem:



holding(A)
clear(B)
onTable(B)

Initial State

Solution: (PutDown(A,B))

on(A,B)

Goal

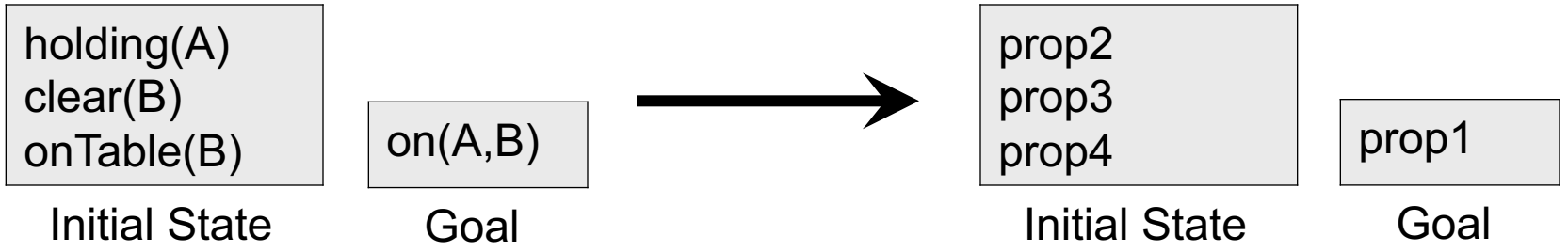
PutDown(A,B):
PRE: { holding(A), clear(B) }
ADD: { on(A,B), handEmpty, clear(A)}
DEL: { holding(A), clear(B) }

PutDown(B,A):
PRE: { holding(B), clear(A) }
ADD: { on(B,A), handEmpty, clear(B) }
DEL: { holding(B), clear(A) }

STRIPS Actions

Propositional Planners

- So far: written propositions (e.g. on(A,B)) in terms of objects (e.g. A and B) and predicates (e.g. on).
- But: planners ignore the internal structure of propositions such as on(A,B).
- Such planners are called **propositional planners** as opposed to first-order or relational planners
- => no difference to the planner if we replace “on(A,B)” in a problem with “prop1” (and so on)
- It feels wrong to ignore the existence of objects. But currently propositional planners are the state-of-the-art.



STRIPS Versus PDDL

- The Planning Domain Description Language (PDDL): standard language for defining planning problems
  - Includes STRIPS as special case along with more advanced features
  - simple additional features: type specification for objects, negated preconditions, conditional add/del effects
  - advanced features: allowing numeric variables and durative actions

- Most planners you can download take PDDL as input
  - Majority only support the simple PDDL features (essentially STRIPS)
  - PDDL syntax is easy to learn from examples packaged with planners

Properties of Planners

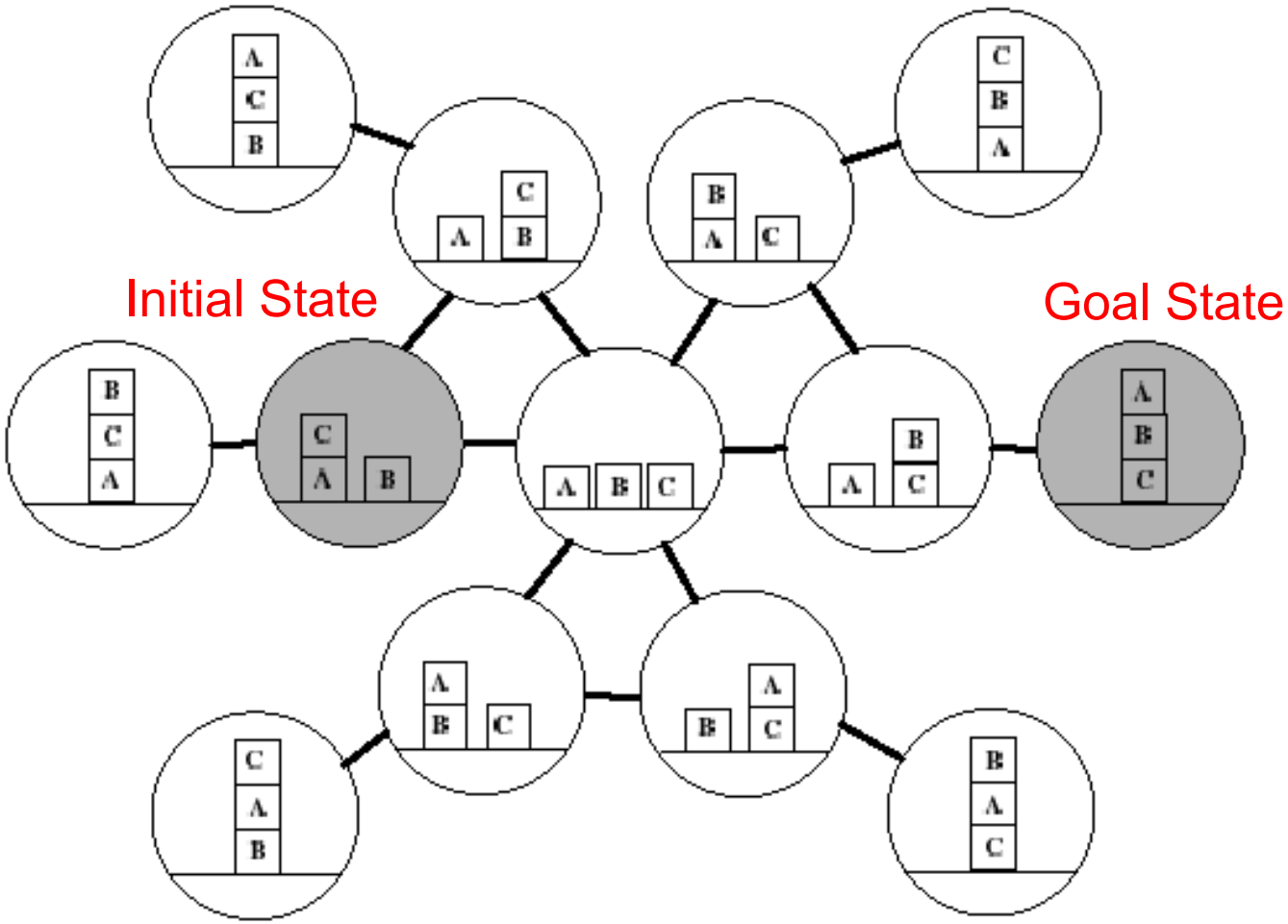
- A planner is **sound** if any action sequence it returns is a true solution
- A planner is **complete** if it outputs an action sequence or “no solution” for any input problem
- A planner is **optimal** if it always returns the shortest possible solution

Planning as Graph Search

- It is easy to view planning as a graph search problem
- Nodes/vertices = possible states
- Directed Arcs = STRIPS actions
- Solution: path from the initial state (i.e. vertex) to one state/vertices that satisfies the goal

Search Space: Blocks World

Graph is finite



Planning as Graph Search

- Planning is just finding a path in a graph
  - Why not just use standard graph algorithms for finding paths?
- **Answer:** graphs are exponentially large in the problem encoding size (i.e. size of STRIPS problems).
  - But, standard algorithms are poly-time in graph size
  - So standard algorithms would require exponential time

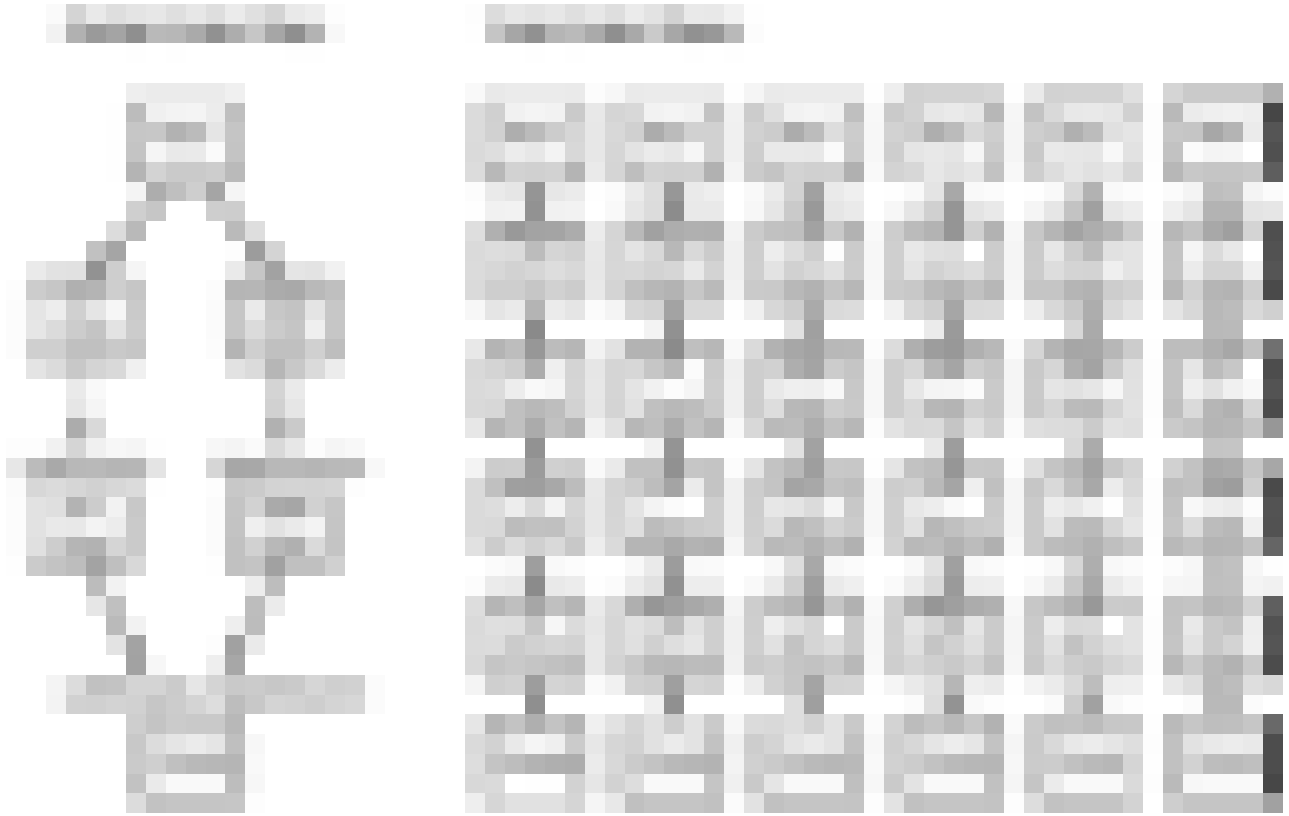
- Do better:
  - We can use A*, but we need an admissible heuristic
    1. Divide-and-conquer: sub-goal independence assumption
    - Problem relaxation by removing
    2. … all preconditions
    3. … all preconditions and negative effects
    4. … negative effects only: Empty-Delete-List

Partial Order Planning (POP)

- State-space search
  - Yields totally ordered plans (linear plans)
- POP
  - Works on subproblems independently, then combines subplans
  - Example
    - Goal(RightShoeOn $\wedge$ LeftShoeOn)
    - Init()
    - *Action*(RightShoe, PRECOND: RightSockOn, EFFECT: RightShoeOn)
    - *Action*(RightSock, EFFECT: RightSockOn)
    - *Action*(LeftShoe, PRECOND: LeftSockOn, EFFECT: LeftShoeOn)
    - *Action*(LeftSock, EFFECT: LeftSockOn)

Berthe Y. Choueiry (Shu-we-ri)
University of Nebraska Lincoln

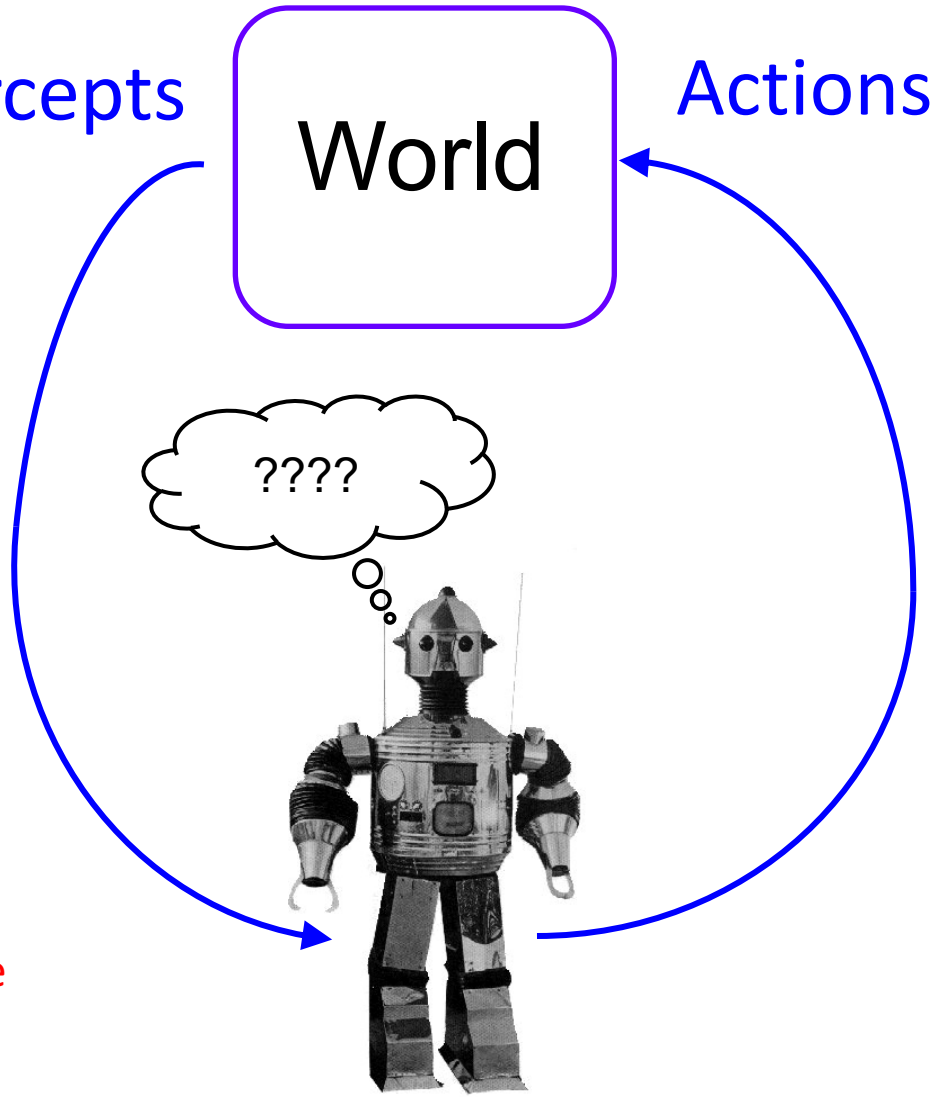
POP Example & its linearization



Berthe Y. Choueiry (Shu-we-ri)
University of Nebraska Lincoln

POMDP

Stochastic/Probabilistic Planning: Markov Decision Process (MDP) Model



perfect

fully observable

sole source of change

stochastic

instantaneous

Goal

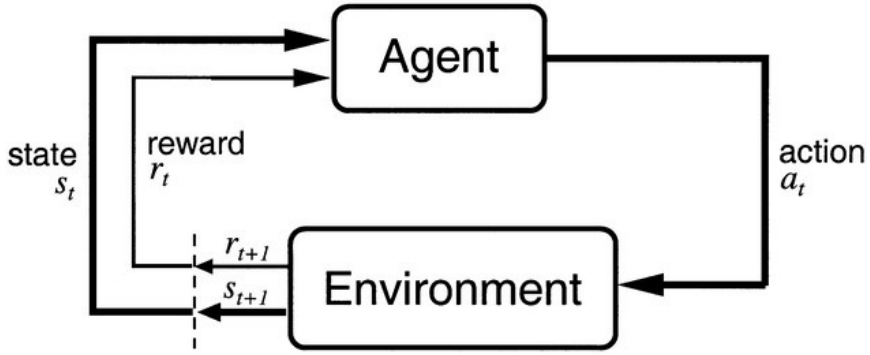
maximize expected reward over lifetime

POMDPs

- A special case of the Markov Decision Process (MDP)
- MDP:
  - Sequential decision making
  - Outcome uncertain
  - the environment is fully observable
  - Markov assumption for the transition model
    - Distribution of next state only depends on current state and action =>
  - the optimal policy depends only on the current state.

- For POMDPs, the environment is only partially observable

Markov Decision Process (S, A, H, T, R)



Given

- S: set of states
- A: set of actions
- H: horizon over which the agent will act
- T Transition dynamics: S x A x S x {0,1,…,H} ➜ [0,1] , $T_t(s,a,s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$
- R Reward: S x A x S x {0, 1, …, H} ➜ $\mathbb{R}$, $R_t(s,a,s')$ = reward for $(s_{t+1} = s', s_t = s, a_t = a)$

Goal: Policy π

- Find : S x {0, 1, …, H} ➜ A that maximizes expected sum of rewards, i.e.,

$$\pi^* = \arg\max_{\pi} \mathrm{E}[\sum_{t=0}^{H} R_t(S_t, A_t, S_{t+1})|\pi]$$

https://ai.stanford.edu/~gwthomas/notes/mdps.pdf

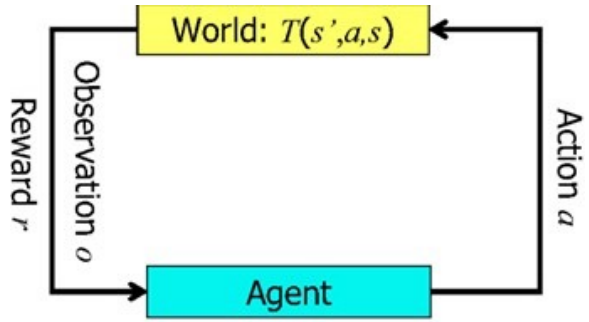
Material: Sachin Patil
Pieter Abbeel, Alex Lee
UC Berkeley

POMDP – Partially Observable MDP

= MDP

BUT

don’t get to observe the state itself, instead get sensory measurements



Now: what action to take given current probability distribution rather than given current state.

POMDPs: Tiger Example

S0
"tiger-left"
Pr(o=TL | S0, listen)=0.85
Pr(o=TR | S1, listen)=0.15

S1
"tiger-right"
Pr(o=TL | S0, listen)=0.15
Pr(o=TR | S1, listen)=0.85



Actions={ 0: listen,
1: open-left,
2: open-right}

Reward Function

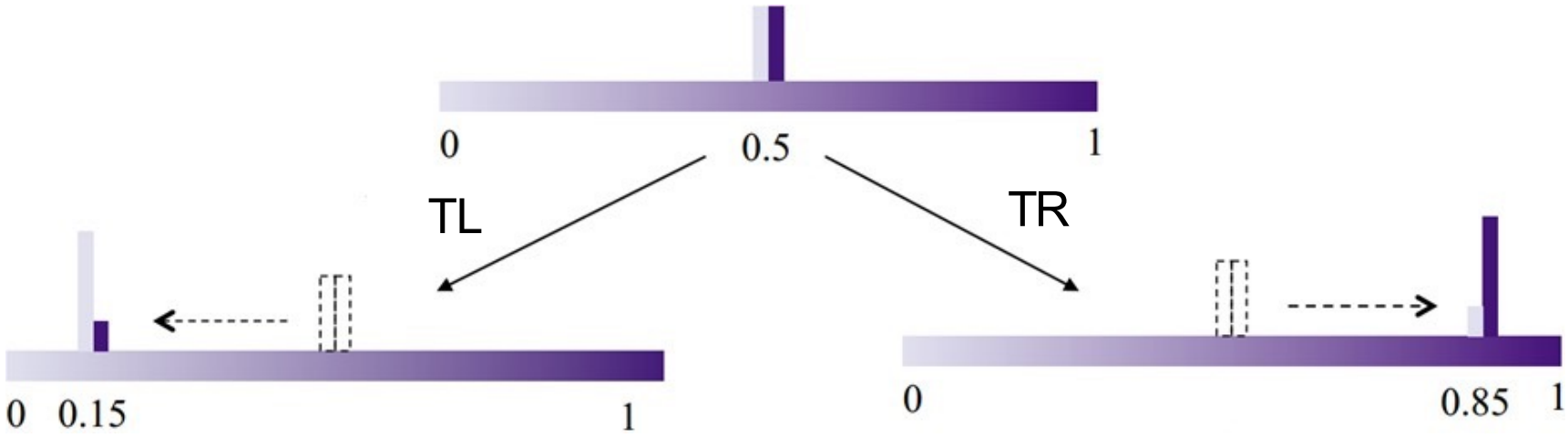
- *Penalty for wrong opening: -100*
- *Reward for correct opening: +10*
- *Cost for listening action: -1*

Observations

- *to hear the tiger on the left (TL)*
- *to hear the tiger on the right(TR)*

Belief State

- Probability of S0 vs S1 being true underlying state
- Initial belief state: p(S0)=p(S1)=0.5
- Upon listening, the belief state should change according to the Bayesian update (filtering)

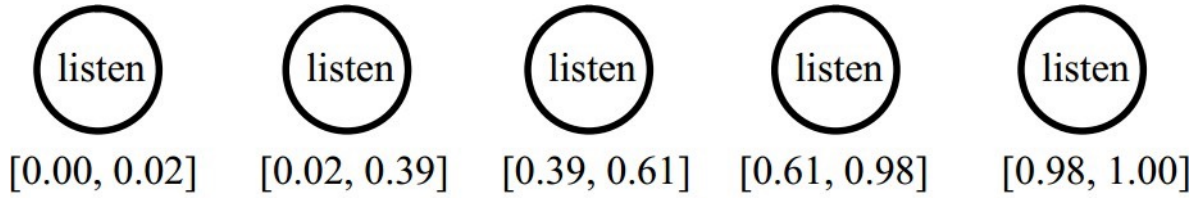


Policy – Tiger Example

- Policy π is a map from [0,1] → {listen, open-left, open-right}
- What should the policy be?
  - Roughly: listen until sure, then open
- But where are the cutoffs?



Tiger example optimal policy for $t = 1$



Tiger example optimal policy for $t = 2$

Some Solution Techniques

- Most exact solution algorithms (value iteration, policy iteration) use dynamic programming techniques
  - transform from one value function (the transition model in physical space, which is piecewise linear and convex - PWLC) to another that can be used in an MDP solution technique
  - Dynamic programming algorithms: one-pass (1971), exhaustive (1982), linear support (1988), witness (1996)
  - Better method – incremental pruning (1996)