64-424
Intelligent Robotics

https://tams.informatik.uni-hamburg.de/lectures/2019ws/vorlesung/ir

Marc Bestmann / Michael Görner / Jianwei Zhang

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Technical Aspects of Multimodal Systems

Winterterm 2019/2020

Outline

1. Decision Making

Decision Making

- In the last lectures we learned how to use sensors to estimate the state of the robot and its environment
- Based on this estimation we need do decide which actions to take to achieve our goals
- This is often called *Decision Making*, *Behavior* or *High-Level Control*
- There are a lot of different approaches to do this, all with their advantages and disadvantages
- As the complexity for the robot's tasks grows, the complexity of the it's decision making grows too
- There is no approach which solves this

There is no silver bullet!

https://arcanum-cyber.com/no-silver-bullets/

No Silver Bullet

- Phrase coined by Fred Brooks in 1986 when talking about software engineering
- "There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity"
- In contrast to advances where we double our computing power every few years (Moore's law)
- The same problem that we already have with general software engineering applies also to robotics

No Silver Bullet

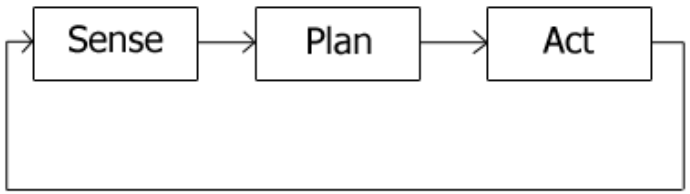
- We can not solve the complexity with a *magic* approach
- There is some *essential complexity* to complex tasks
  - If a robot has to do 3 different tasks, then you can't get around programming or teaching these 3 tasks somehow
- But there is also *accidental complexity*
  - Complexity coming from how the system is implemented
- We can solve accidental complexity by using better methods
  - using a high-level programming language like Python instead of machine code
  - using programming paradigms
  - or in robotics by using a fitting control architecture

Robotic Paradigm

- We can hardly use numbers to say what is the best approach (quantitative approach)
- But we can use reasoning to get some insights in a inductive way (qualitative approach)
- Before talking about concrete approaches for decision making, we should talk about categories
- There are four classes of robot control methods:
  - Deliberative
  - Reactive
  - Hybrid
  - Behavior-Based

Deliberative

- "Think, Then Act"
- Top-down approach
- Sensing, filtering, modeling, planning, execution
- Highly sequential
- Complex behaviors easy to model
- Can plan into the future by predicting results of its actions



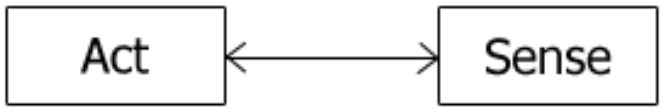
https://en.wikipedia.org/wiki/Robotic_paradigm
Springer Handbook of Robotics, Chapter 13

Deliberative

- In most real life scenarios almost impossible to use, due to noise and unforeseen changes
- Applications tend to have a global world model
- Planning tends to take time, robot is not very reactive
- Examples:
  - Planning the complete path of a wheeled robot using a model of the world
  - Plan multiple arm movements to cook something

Reactive

- "Don't Think, (Re)Act"
- Bottom-up approach
- Multiple sense-act couplings
- Higher level behaviors emerge implicitly
- Very fast reaction time, due to no planning
- Insects are largely working reactive



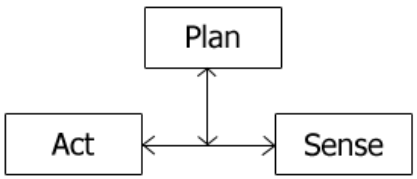
https://en.wikipedia.org/wiki/Robotic_paradigm
Springer Handbook of Robotics, Chapter 13

Reactive

- Capable of optimal performance for some problem types
- Not usable if internal model, memory or learning is required
- Examples:
  - Navigating a robot purely based on its current sensor inputs and a general goal direction

Hybrid

- "Think and Act Concurrently"
- Tries to combine the best of both worlds
- Deliberative part
  - Plans long term goals (low update rate)
  - Guides reactive part towards more optimal trajectories and goals
- Reactive part
  - Deals with current changes (high update rate)
  - Override deliberative part if unforeseen changes happen



https://en.wikipedia.org/wiki/Robotic_paradigm ; Springer Handbook of Robotics, Chapter 13

Hybrid

- Also called *layered robot control*
- Examples:
  - Path planning with additional reactive system for avoidance of obstacles
  - Humanoid robot with reactive system for reflexes (falling, standing up) and deliberative system for its high level goal

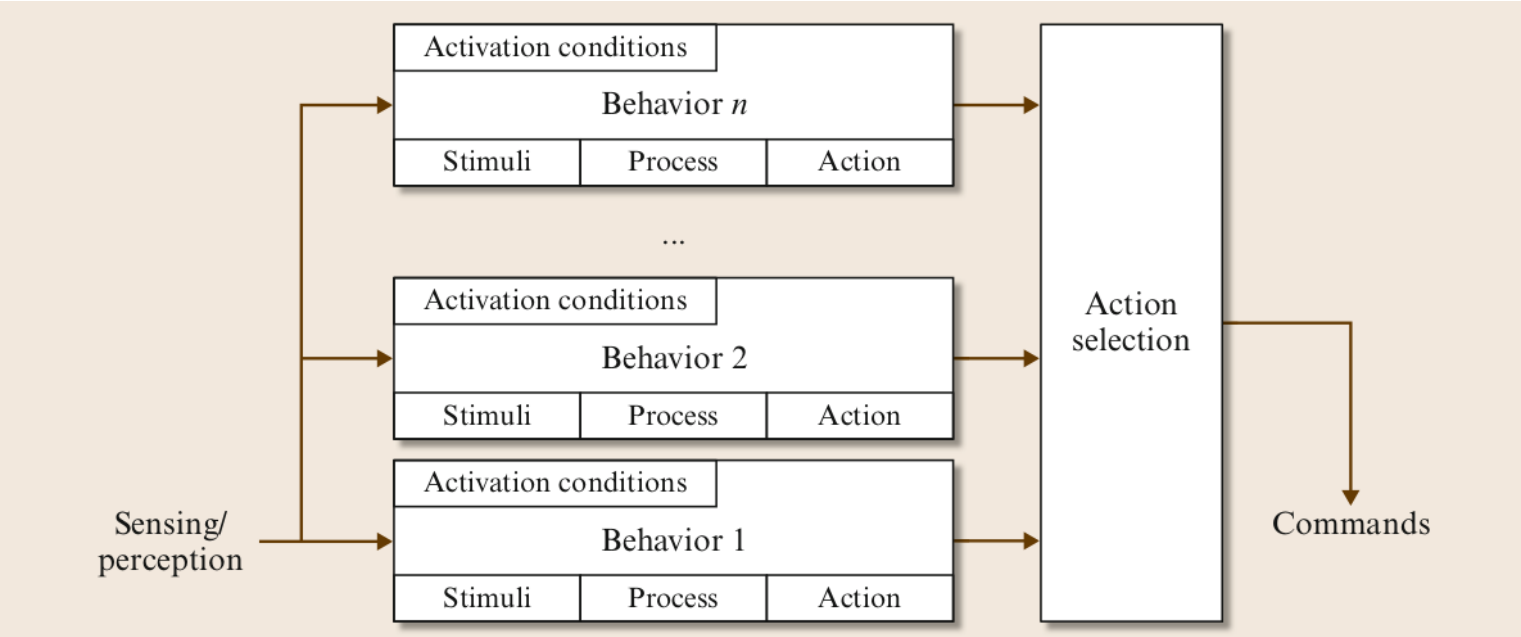
Behavior-Based

- "Think the Way You Act"
- Bottom-Up (similar to reactive approach)
- Multiple distributed, interacting modules, called behaviors
- Each behavior has its own goal
- Over all system behavior emerges from the behaviors
- No centralized world model, but every behavior has its own model

https://en.wikipedia.org/wiki/Robotic_paradigm

Springer Handbook of Robotics, Chapter 13

Behavior-Based



Springer Handbook of Robotics, Chapter 13

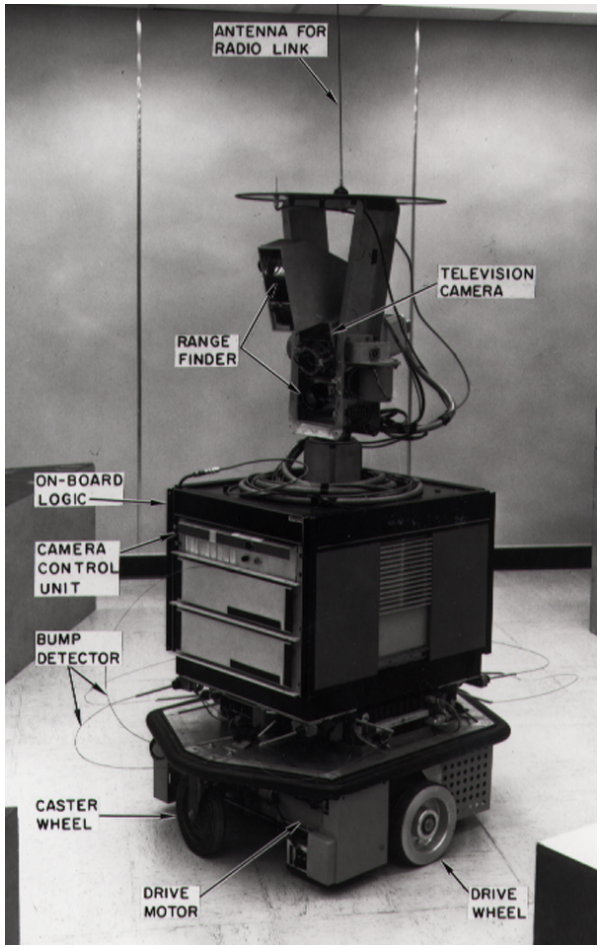
What we will discuss in the next lectures

- Classical approaches
  - Symbolic reasoning (1950s)
  - Fuzzy logic (1965)
  - Subsumption (1986)
  - Decision trees (<=1963)
  - Finite state machines (<=1962)
  - Hierarchical finite state machines (state charts) (1980s)
- Recent approaches
  - Behavior trees (~2001)
  - Dynamic Stack Decider (2018)
- Design principals
- Advantages and disadvantages
- What to use when

Summary

Motivation

- Symbolic reasoning is one of the oldest and best understood subfields of Artificial Intelligence
- Also called GOFAI ("Good Old-Fashioned Artificial Intelligence")
- Autonomous robots should act intelligently on low *and high* levels of abstraction
- Real-world tasks are usually given on an abstract level in



SRI's Shakey

Examples of Symbolic Tasks for Autonomous Robots

Often, robots should exhibit intelligent adaptive "higher-level" behavior, with respect to their environment.

- Unblock the doorway (Shakey, 1966)
  - It is not specified how to get there
- Fetch a sandwich (JSK Tokyo, 2011)
  - Fetch it from the fridge or buy it in the store next-door
- Prepare a pancake (TUM/IAI Bremen, 2010)
- Set the table (Project RACE, TAMS, Hamburg)
- ...

These tasks require background theories and reasoning capabilities.

Symbolic vs. Statistic Reasoning

- symbolic (e.g. STRIPS):
  - Using symbols to represent the world and reason on it
  - Explainable
  - Needs not much data
  - Examples tasks:
    - planning
    - reasoning
    - language generation (sentences)
- statistical (e.g. something using a neural network):
  - Use statistics to represent the world and reason on it
  - Can handle uncertainty
  - Gets better with more data
  - Examples tasks:
    - pattern recognition
    - speech generation (sound)
    - motion skills (e.g. walking)

Symbols

A symbol is a single token (or "atom") that can be uniquely identified among others.

- Symbols by themselves are syntactic entities
- Algorithms can operate on them
- They do not carry an inherent meaning/semantics
- Symbols in everyday life are associated with intuitive semantics, e.g. "tree", "red", "to shake"
- Science aims to give precise/consistent definitions of all symbols in use, e.g. "$\mathbb{N}$", "$+$", "$\varnothing$"

Assignable Semantics

In symbolic modeling the semantics of each symbol are influenced by

- Human interpretation
  - What does the designer/the user assume the symbol means?
- The relationship to other symbols
  - e.g. encoded as a theory in first-order predicate logic
- The anchoring/grounding of the symbol in the whole system
  - How does the occurrence of the symbol influence the system?

When I use a word, it means just what I choose it to mean—neither more nor less. – by Lewis Carroll

Human Concepts and Formal Symbols

There is a large discrepancy between formal symbols and everyday human concepts:

- Human concepts are backed up by commonsense knowledge, formal symbols rely on the specified theory
- Human concepts are intrinsically vague, this is hard to model for symbols *in a compatible way*
- Human concepts are often polysemous or homonymous when mapped onto engineered symbols

- These differences can break the autonomous behavior and user interaction in unforeseen situations

Polysemous Examples

- Polysemous (different meaning but a semantic relation)
  - Mouse
    - A small rodent
    - A digital input device
  - Door
    - the object which swings open to allow entrance, as in "Open the door."
    - the opening created thereby, as in "Walk through the door."
- Homonymous (no semantic relation)
  - Bass
    - Type of fish
    - A tone of low frequency
    - A musical instrument

Symbol Grounding

The connection between symbols used inside an agent and their counterparts in the real world is called *grounding*.

This has to rely on the robot's sensing and performance capabilities

- Perception modules can signal internal symbols by interpretation of sensor data
- Action modules can control the robot's behavior whenever special symbols are inferred

Symbol Grounding (2)

Perception modules

- Subfields of computer vision focus on individual modules
- Limited by the quality of the sensor data and applied perception algorithms
- Examples: `in_front_of(table1)`, `in(red_box, room2)`, `filled(bottle, 0.5)`, `activity(harry, reading)`

Action modules

- Focus of traditional robotics
- Control theory, reinforcement learning, ...
- Limited by motor accuracy, proprioception, applied control algorithms, *perception*
- Examples: `move(position5)`, `place(object1, table2)`, `open(shelf5)`, `plug_in_to_outlet(o1)`

On the Attribution of Semantics

- To get to a running behavior *all* modules have to be implemented in a consistent way.
- Many are active research fields

$\Rightarrow$ Most modules rely on simplifying assumptions / stubs

- These drastically reduce the scope and still capture human expectations in the prepared demo setup
- *But*, they often lead to unintuitive behavior and strangely prepared demo setups

- Next time you see a robotic demonstration, you might ask:
  - Which assumptions did the researchers have to make?
  - How many modules would have to be drastically changed to get this to work in a different environment?

Symbol Grounding (3)

Many symbols represent aspects of the environment that are difficult to ground

- Fill level of an opaque bottle, intention of a human, ...
- Try to infer them via explicit perception actions, dedicated prediction modules

Many action symbols can be defined in terms of other symbols and don't have to be implemented directly:

- `do_the_laundry`, `prepare_a_pancake`, `set_table(table2)`
- They can be grounded in terms of sequences of more basic actions

Primitive Actions

- Actions can either be implemented by modules or defined through other actions
- This leaves the choice of primitive/atomic actions
- Primitives have to have well-defined outcomes
- The set of primitive actions should support other demonstrations in the same environment
- Obvious candidates are
  - `set joint to position` X
  - `move base to position` X
  - `look at` X
  - `pickup` X
- Nevertheless, `do_the_laundry` can be a justified primitive action, assuming it "does the job"

Blocks World Planning

- Ignoring grounding, the set of relevant symbols can be used for symbolic inference
- Problems without grounding are sometimes referred to as "blocks world" problems, alluding to the blocks world domain
- Here, everything behaves as specified *by definition*

A Word On Notation

Classical predicate logic represents

- atomic statements as "`predicate(param1, param2)`"
- variables as capital letters "`X`, `Y`, `Z`"
- e.g. "`on(glass, X)`"

A large part of the planning community uses an equivalent Lisp-like syntax instead:

- statements: "`(predicate parameter1 parameter2)`"
- variables: "`?var1`"
- e.g. "`(on glass ?place)`"

STRIPS Planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- plans purposeful sequences of actions in blocks world domains
- defined a de-facto standard form for the formulation of planning problems

STRIPS planning domains contain

- a set of (boolean) variables, i.e. *fluents*, with mutable values
- discrete states characterized by their valid fluents
- a set of action specifications
- actions mark state transitions

STRIPS Planning (2)

STRIPS problems consist of

- The set of relevant fluents $\mathcal{F}$
- The set of effective action schemas $\mathcal{O}$
- The initial set of true fluents $\mathcal{I}$
- The set of literals (fluents or their negation) to be satisfied $\mathcal{G}$

Action schemas consist of

- a *signature* including all free variables, e.g. `action(P1,P2)`
- a set of literals as *precondition* for the action
- a set of literals as *effects* of the action

The STRIPS Blocks World

$\mathcal{F}$: $X \in \{a, b, c\}, Y \in \{a, b, c, table\}$

X is on top of Y...	$On(X, Y)$
There is nothing on top of X...	$Clear(X)$
X is a movable block...	$Block(X)$

$\mathcal{O}$:

- `move(B,X,Y)`
  - preconditions: $\{On(B, X), Block(B), Block(Y), Clear(B), Clear(Y), B \neq X, B \neq Y, X \neq Y\}$
  - effects: $\{On(B, Y), \neg On(B, X), Clear(X), \neg Clear(Y)\}$
- `move_to_table(B,X)`
  - preconditions:$\{On(B, X), Clear(B), Block(B), Block(X), B \neq X\}$
  - effects: $\{On(B, table), \neg On(B, X), Clear(X)\}$

The STRIPS Blocks World

Assume this initial state and goal:

$$\begin{aligned}\mathcal{I} =& \{Block(a), Block(b), Block(c), Clear(b), Clear(c)\} \\ & \cup \{On(c, a), On(b, table), On(a, table)\} \\ \mathcal{G} =& \{On(a, b), On(b, c)\}\end{aligned}$$

Given $\langle \mathcal{F}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, a planner can find a sequence of variable-free and applicable actions, i.e. a **plan**, to achieve $\mathcal{G}$ from state $\mathcal{I}$

The STRIPS Blocks World

Assume this initial state and goal:

$$\begin{aligned}\mathcal{I} =& \{Block(a), Block(b), Block(c), Clear(b), Clear(c)\} \\ & \cup \{On(c, a), On(b, table), On(a, table)\} \\ \mathcal{G} =& \{On(a, b), On(b, c)\}\end{aligned}$$

Given $\langle \mathcal{F}, \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, a planner can find a sequence of variable-free and applicable actions, i.e. a **plan**, to achieve $\mathcal{G}$ from state $\mathcal{I}$

One plan for this problem is

- `move_to_table(c,a)`, `move(b,table,c)`, `move(a,table,b)`

Open and Closed Worlds

- STRIPS assumes full information on the fluents of each state
- If a fluent is not in the state description, it is *false*
- This is called the **Closed World Assumption**
- In practice, inference schemes often rely on this assumption

. . . but just because you don't know your car was stolen, that does not mean it is still where you parked it.

- Many planners do not make the CWA, but the resulting inferences are much weaker

. . . just because you parked your car somewhere, that does not mean it is there anymore.

PDDL

- Planning competitions (e.g. IPC) emerged
- . . . together with the need for a standardized language
- AIPS-98 introduced the **P**lanning **D**omain **D**efinition **L**anguage
- Provides means to specify STRIPS problems
- By now, supports many more elaborate constructions:
  - numeric fluents
  - continuous actions
  - timed events
  - preferences
  - function symbols (object-fluents)

PDDL - an Example Domain

```
(define (domain gripper-strips)
(:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r)
             (at ?b ?r) (free ?g) (carry ?o ?g))
(:action move
:parameters (?from ?to)
:precondition (and (room ?from) (room ?to) (at-robby ?from))
:effect (and (at-robby ?to) (not (at-robby ?from))))
(:action pick
:parameters (?obj ?room ?gripper)
:precondition (and (ball ?obj) (room ?room)
                   (gripper ?gripper) (at ?obj ?room)
                   (at-robby ?room) (free ?gripper))
:effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
             (not (free ?gripper)))))
```

PDDL - an Example Problem

```
(define (problem strips-gripper2)
(:domain gripper-strips)
(:objects rooma roomb ball1 ball2 left right)
(:init (room rooma)
       (room roomb)
       (ball ball1)
       (ball ball2)
       (gripper left)
       (gripper right)
       (at-robby rooma)
       (free left)
       (free right)
       (at ball1 rooma)
       (at ball2 rooma))
(:goal (at ball1 roomb)))
```

Modeling A Domain

What are useful fluents/operators for a robot that has to deliver packages in a building?

Modeling Knowledge

- Autonomous robots often preform in similar environments
- For a start, all perform in the mundane world, where
  - Temporal ordering is transitive
  - People cannot remember things that will happen in the future
  - If you take a rigid object away, it will not be where it was anymore
  - If you cut up cheese, you will get smaller blocks of cheese
  - But if you cut up a cup, you will end up with shards
- This kind of commonsense knowledge has to be hand-crafted into each individual planning problem!

$\Rightarrow$ Collect such knowledge in a background theory, i.e. **ontology**

Modeling Common Sense Knowledge

- There have been many such projects in the past
- Prominent historical example:
  Patrick Hayes: "*Naive Physics I: Ontology for Liquids*" - 1978

Ongoing projects:

- (Open)Cyc / RoboEarth
  - Carefully handcrafted ontologies
  - Let's have a look inside...
- "Semantic Web"
  - Ongoing attempt to distribute knowledge engineering
  - Incompatible / conflicting ontologies

Modeling Common Sense Knowledge (2)

There are limits to what can be modeled

- A lot of commonsense knowledge is difficult to describe in boolean symbols
- Modeling probabilities/fuzzy logic leaves the problem where to get consistent numbers and makes inference much harder

- Adding background theories makes inference arbitrarily difficult
- Full predicate logic is *undecidable*
- There are various syntactic restrictions with different runtime complexity and expressiveness

RDF and OWL

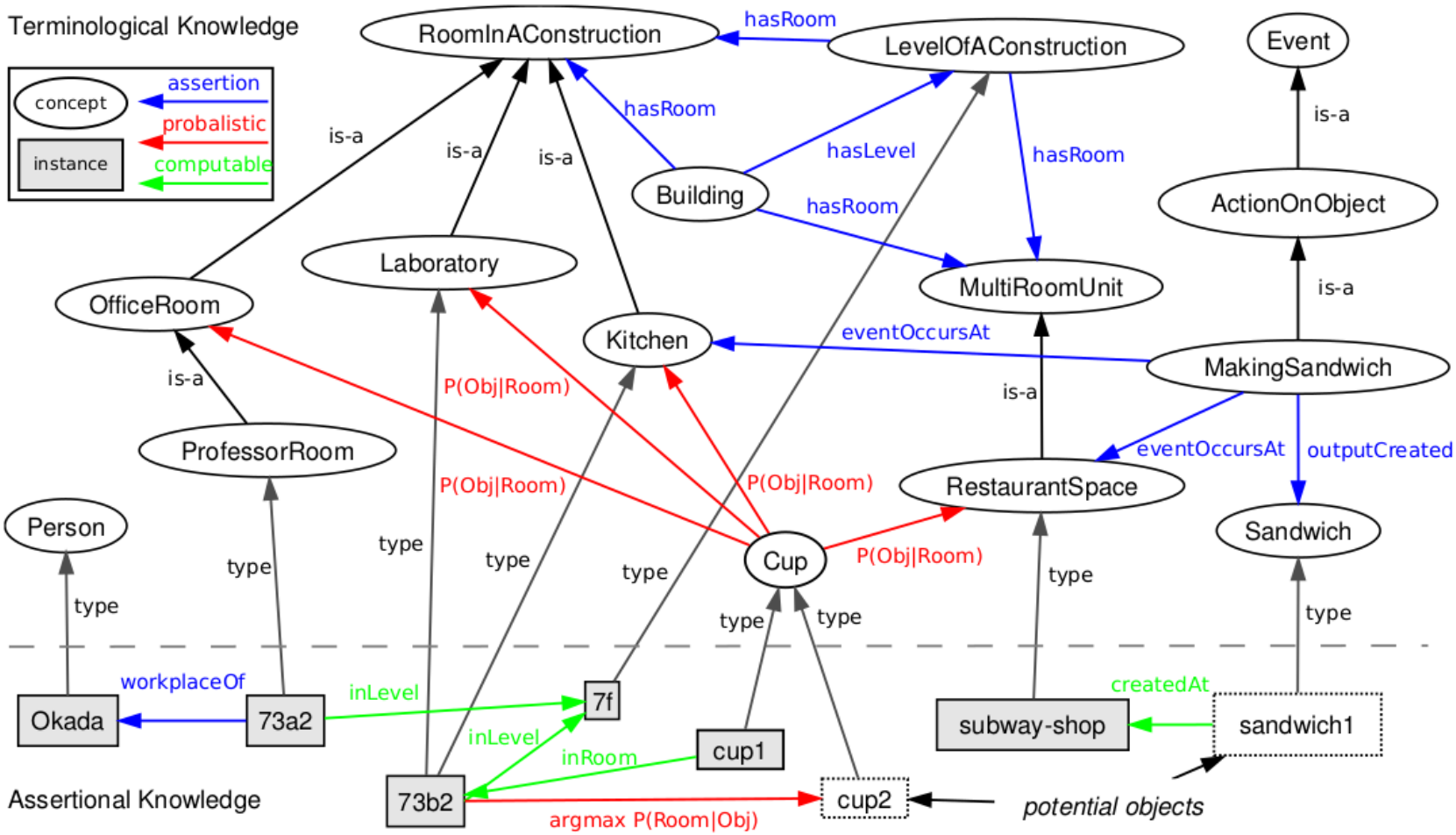
W3C standardized ontology notation on the web

- The **R**esource **D**escription **F**ramework
  - XML-based storage format for symbolic graphs
  - Based on triples expressing relations

```
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#fullName>
"Eric Miller"
```

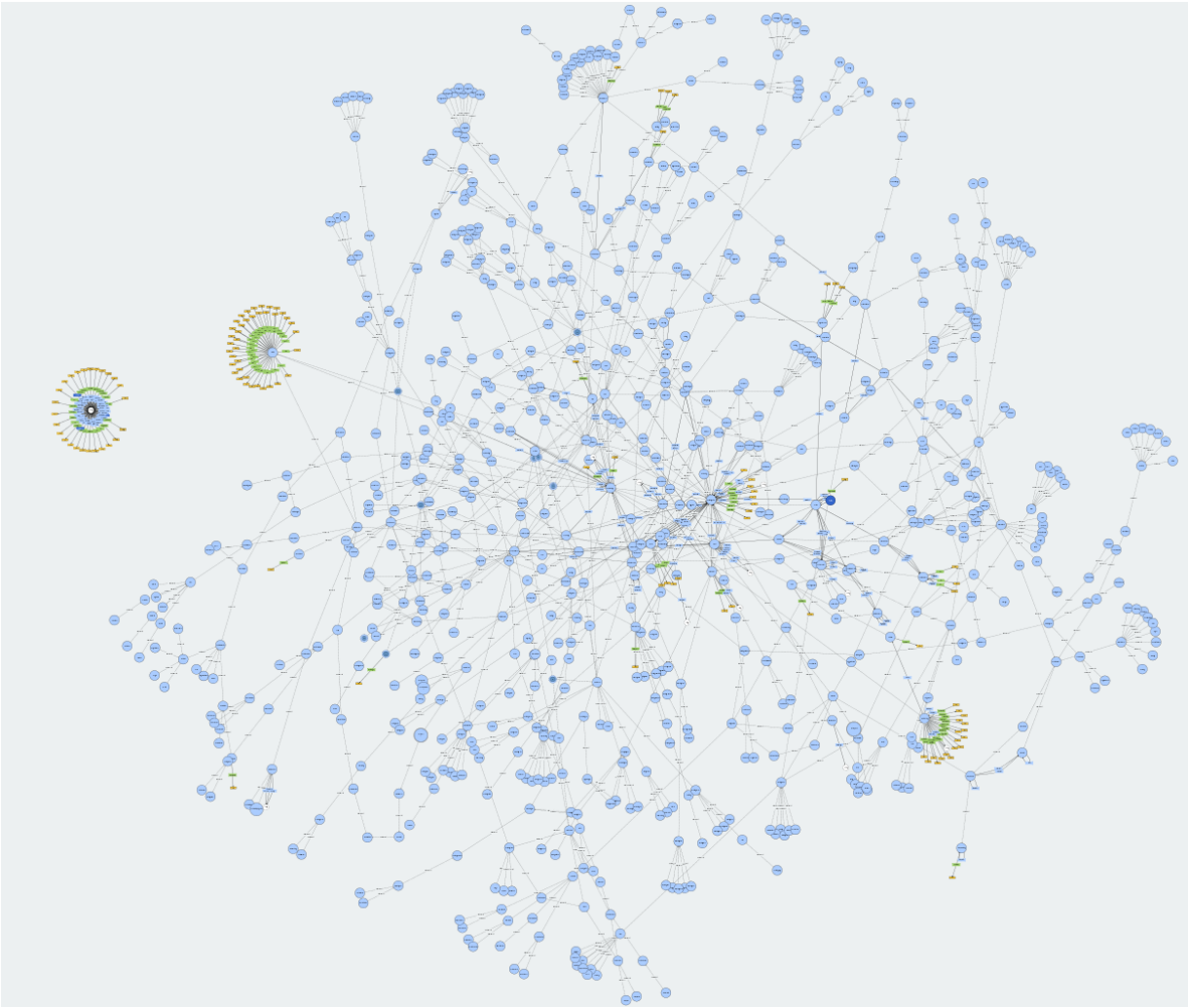
- The **W**eb **O**ntology **L**anguage (OWL)
  - Builds on RDF
  - Most prominent version: **OWL DL** implements formal Description Logic (DL)
  - supported by many tools / reasoners

Ontologies in the Wild - Fetching Sandwiches



Ontologies in the Wild - RACE

Ontologies in the Wild - KnowRob



Design Principles

- There are some design principles (similar to software architecture) for high-level control
- We can use them to investigate their pros and cons
- Some use cases require our

Design Principles of Control Architectures (CA)

- Hierarchical organization
  - Some subtask may be more important than others
  - Ex: recharging when empty > navigating to goal
- Reusable code
  - The same subtask is maybe needed multiple times
  - Ex: turning sensors to specific location
- Modular design
  - Splitting a task into subtasks makes development easier
  - Ex: divide "grasp" into "open hand", "position hand", "close"
- Maintainability
  - Changes to the behavior has to possible without general restructuring
  - Ex: adding "lift hand" to "grasp" should only require changes in this part

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017
Poppinga, Martin and Bestmann, Marc. "ASDS - Active Self Deciding Stack", 2018

Design Principles of Control Architectures (CA) (cont.)

- Human readable
  - Structure has to be readable for developing and debugging
  - Ex: GUI with graph structure and current state
- Stateful
  - The current state of the system should be clear
  - Ex: clear if ball is currently in hand or not
- Fast
  - Low latency between sensor input and action
  - Ex: Bumper is hit -> immediate stop of wheels to prevent damage
- Expressive / scalable
  - CA must be able to encode a large variety of tasks
  - Ex: a soccer player with different strategies

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017
Poppinga, Martin and Bestmann, Marc. "ASDS - Active Self Deciding Stack", 2018

Design Principles of Control Architectures (CA) (cont.)

- Suitable for automatic synthesis
  - Synthesis, e.g. by machine learning, for action ordering
  - Ex: using NNs in some parts to decide which action is to be taken
- Understandability of the concept
  - It should not take to much time to understand the concept
  - Ex: FSM is very simple, BT is complex
- Implementation effort
  - Effort to implement the used concept
  - Not important if fitting library is available

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017
Poppinga, Martin and Bestmann, Marc. "ASDS - Active Self Deciding Stack", 2018

Design Principles

- A lot of things to keep in mind
- Sometimes contradictory
  - Ex: fast <-> expressive
  - EX: maintainability <-> understandability
- Highly depended on the domain and goal
- Keep in mind: there is no silver bullet!

Symbolic Reasoning

- Hierachical org.: Bad
  - Multiple goals possible but same importance
- Reusable: Good (ontologies)
- Modular: Very good
- Maintain.: depends
  - Actions very maintainable, state not
- Human read.: Good
- Stateful: Very good
- Fast: Bad
  - The classic example of deliberative approach
- Expressive: Good
- Synthesis: ?
- Understandable: Good
- Effort: Bad

Decision Trees (DT)
Behavior Trees (BT)
Dynamic Stack Decider (DSD)
Summary

Introduction

The concept of fuzzy logic was introduced by Lotfi Zadeh (1965)

- Inspired by human information processing capabilities
- People do not require precise, numerical information input, yet they are capable of highly adaptive control

General assumption

- If feedback controllers could be programmed to accept noisy, imprecise input, they would be much more effective and perhaps easier to implement
- How important is it to be exactly right when a rough answer will do?

Fuzzy control (cont.)

- *Fuzzy* means: blurred, diffuse, vague, uncertain, ...
- Fuzzy control uses fuzzy sets as mechanism for
  - Abstraction of unnecessary or too complex details
  - Troubleshooting of problems which are not easily solvable by a simple *yes* or *no* decision
  - Modeling of (*soft*) concepts without any sharp borders

Fuzzy control (cont.)

- Unsharp linguistic grading of terms like **"big"**, **"beautiful"**, **"strong"** ...
- Human thinking models and behavior models with first-level logic
  - **Car driving:** *if-then*-rules
  - **Car parking:** Accurate up to a milimeter?
- Fuzzy speech instead of numerical description
  - "Brake 2.52 m ahead of the curve!" $\rightarrow$ only in machine systems
  - "Brake shortly before the curve!" $\rightarrow$ in natural language

Adaptive control methods

Fuzzy control is generally a good fit for realization of *adaptive control methods*

- Control can be understood as mapping from a sensor space onto actions
- In many cases it is *a priori* unknown, which measurement parameters are especially important for the choice of actions
- Some systems are very hard to describe in a mathematical way
- Often, sensor data is inaccurate, noisy and/or high dimensional

Adaptive control methods (cont.)

Models for adaptive control systems

- The creation of an ideal mapping between sensor space and actions is very difficult with classical methods of control engineering
- In order to control such systems, a simpler method needs to be used for description
- Neural networks
- Fuzzy-Controller

Linguistic variables

Linguistic variables are one of the main building blocks of fuzzy logic

- A *linguistic variable* is a variable, which can take on a range of linguistic terms
- A *linguistic term (value, label)* is the quantification of a term from natural language through a fuzzy set
- Many terms of natural language can be characterized through degree of membership related to fuzzy sets
- Therefore, fuzzy sets can be considered as the basic tool for modelling of *linguistic terms*

Linguistic variables (cont.)

Examples:

- Linguistic variable: "**SPEED**"
- Linguistic terms of "**SPEED**"
  "**high**", "**low**", "**rapid**", "**economical**"

- Linguistic variable: "**BUILDING**"
- Linguistic terms of "**BUILDING**"
  "**cottage**", "**bungalow**", "**skyscraper**"

Characteristic function

Crisp sets can be defined through specification of their characteristic function:

$$\mu_{\mathcal{A}}(x) = \begin{cases} 1 & \text{for } x \in \mathcal{A} \\ 0 & \text{for } x \notin \mathcal{A}, \end{cases}$$

where $\mu_{\mathcal{A}} : X \rightarrow \{0, 1\}$
Example:
"Apple" would be a fruit (result 1) but "Potato" not (result 0).

Membership function

For **fuzzy sets** A, a generalized characteristic function μ_A is used, which maps a real number $[0, 1]$ to each element $x \in X$:

$$\mu_A : X \rightarrow [0, 1]$$

- The function μ_A is called membership function (MF)
- It indicates the "degree", to which the element x belongs to the described unsharp set A ($\rightarrow$ fuzzy set)
- Example:
  - "Apple" would be 1 for the set "Fruit"
  - "Potato" would maybe be 0.1 for the set "Fruit" since it is not really a fruit but closer to it than e.g. "Car"

Membership function (cont.)

Representation of membership functions

- Discrete representation
  - Fixed-size array
  - Saving of the MF-Values for the whole x-codomain
- Parametric representation
  - Functions with parameters (less space required)
  - Typical types: Singleton, triangular shape, trapezoid shape, bell curve, B-Spline basis function

Membership function (cont.)

Creation of the membership functions

- Context-dependent specification
  - Experimental, domain- and application specific
- Construction using sample data
  - Clustering
  - Interpolation
  - Curve Fitting (Least-squares)
  - Neural networks
- Knowledge acquisition through experts
  - One or several experts
  - Directly and indirectly

Fuzzy set

A **fuzzy set** A over a universe X is given through a mapping $\mu_A : X \to [0, 1]$.

For all $x \in X$, $\mu_A(x)$ denotes the degree of affinity (membership) of x in A

Example:

- The set of integer numbers approximately equal to 10:

$$A_{10} = (0.1, 7), (0.5, 8), (0.8, 9), (1.0, 10), (0.8, 11), (0.5, 12), (0.1, 13)$$

Fuzzy control

In a fuzzy control system, the influence on dynamic circumstances of a fuzzy system is characterized by a set of linguistic description rules

IF (a set of conditions is satisfied)
THEN (a set of consequences can be infered)

Conditions (antecedents or premises) of the IF-part:

→ Linguistic variables from the domain of process states

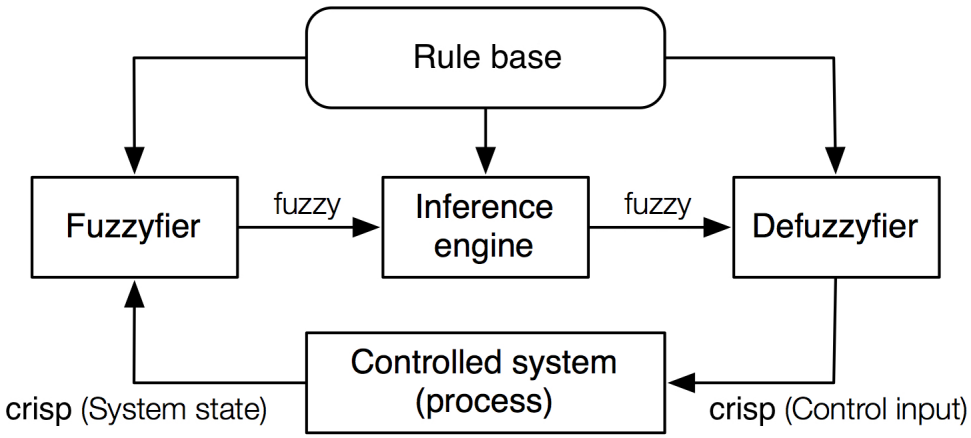
Conclusions (consequences) of the THEN-part:

→ Linguistic variables from the control domain

Components of fuzzy control

A complete fuzzy controller consists of four major components

- Rule base
- Fuzzifier
- Inference engine
- Defuzzifier



Rule base

The rule base is the central component that stores expert knowledge

- It contains the control strategies in the form of IF-THEN rules

The rule base is considered a part of the knowledge base of the fuzzy controller, other components being:

- The input membership functions
  $\rightarrow$ required for *fuzzification* of crisp input values
- The output membership functions
  $\rightarrow$ required for *defuzzification* of the inference result(s)

Rule base (cont.)

Example:

Given a fuzzy control system with two inputs A and B and a single output C, the general representation of the rule base would be:

R_1: IF (x is A_1 OR y is B_1) THEN (z is C_1)

R_2: IF (x is A_2 OR y is B_2) THEN (z is C_2)

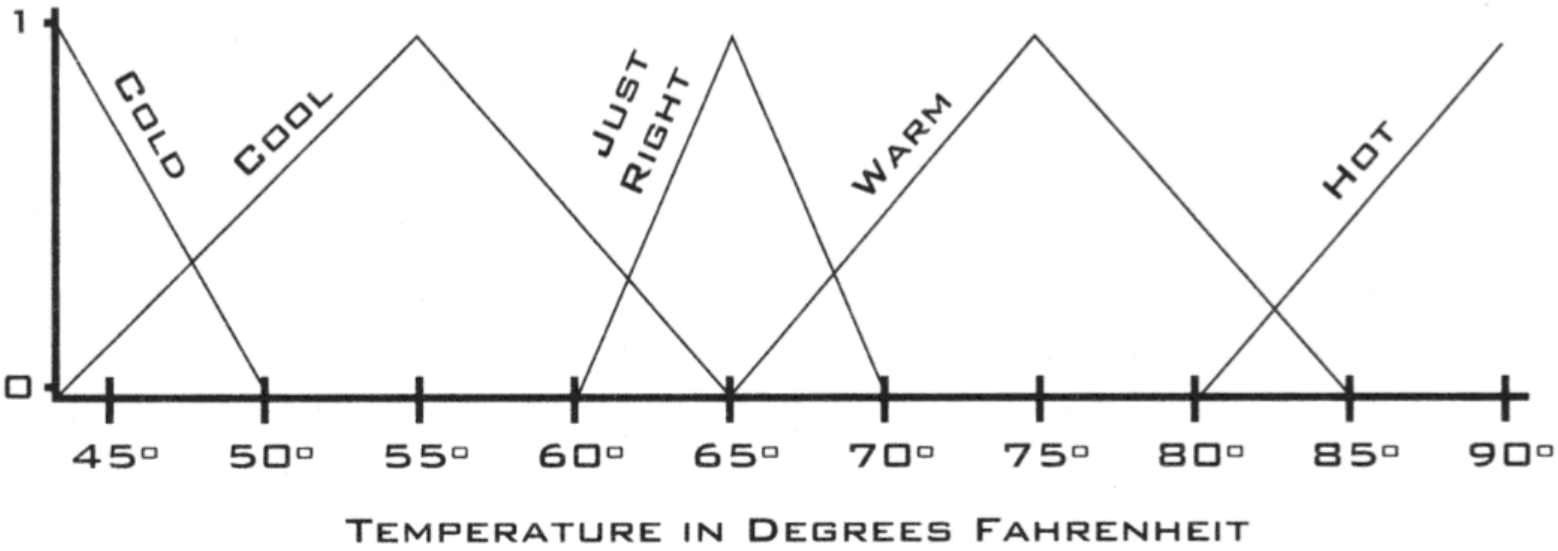
...

R_k: IF (x is A_k OR y is B_k) THEN (z is C_k)

Note: Inputs and outputs mentioned here are strictly fuzzy.

Fuzzifier

The fuzzifier converts the crisp input values into fuzzy representations based on the membership degree to established fuzzy sets



Fuzzifier (cont.)

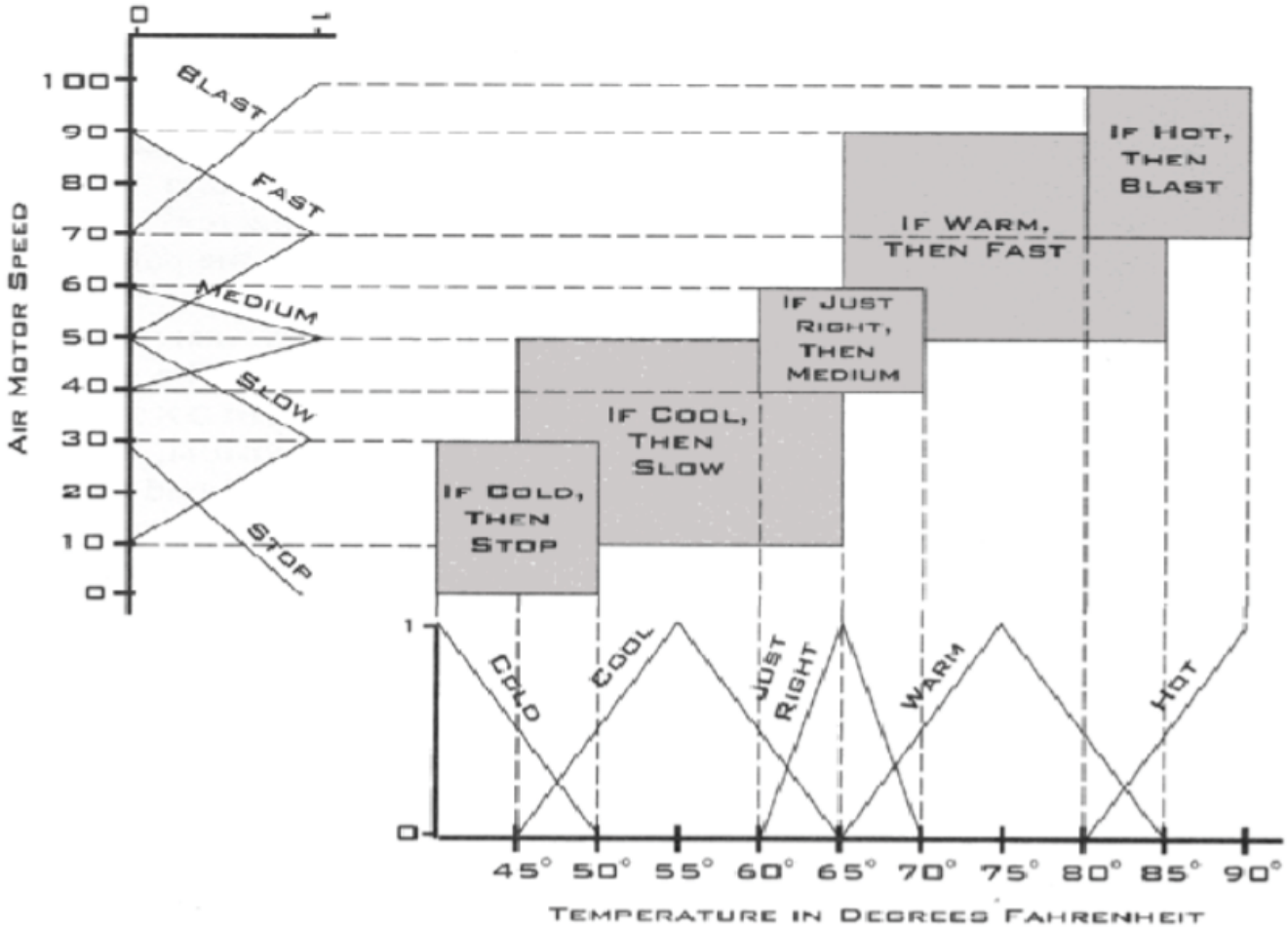
- The established fuzzy sets and associated membership functions are designed to exploit the inherent inaccuracy of input data (e.g. from sensors)
- The fuzzifier approximates the human reasoning process
- While expert knowledge of the process to be controlled is helpful, it is not a requirement
- The overall implementation effort of the controller is reduced significantly

Inference engine

The inference engine processes the *fuzzified* input values based on evaluation of the rule base

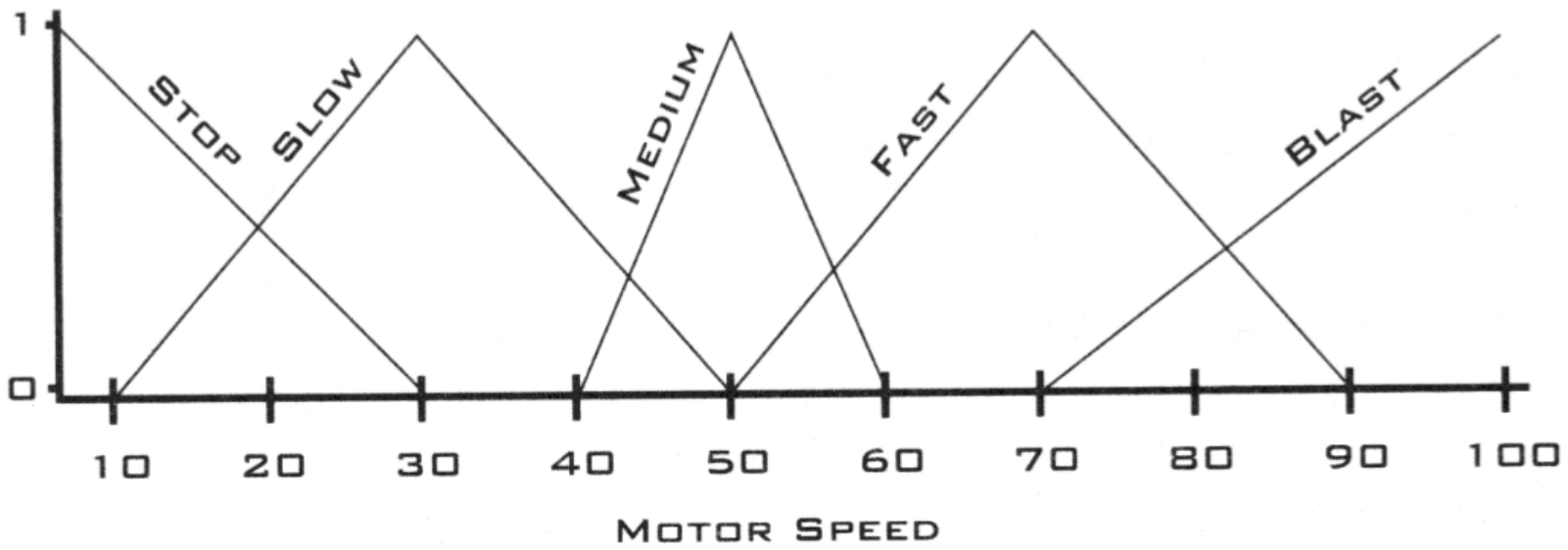
- *All* rules within the rule base are evaluated
- The evaluation usually occurs in parallel (depends on implementation)
- All evaluated rules contribute to the fuzzy output value to some degree
- The contribution of most rules to the output value is 0
- The resulting fuzzy output value is a *union* of the results of all evaluated rules

Inference engine (cont.)



Defuzzifier

The defuzzifier converts the obtained fuzzy output value into a crisp representation based on the output membership functions



Defuzzifier (cont.)

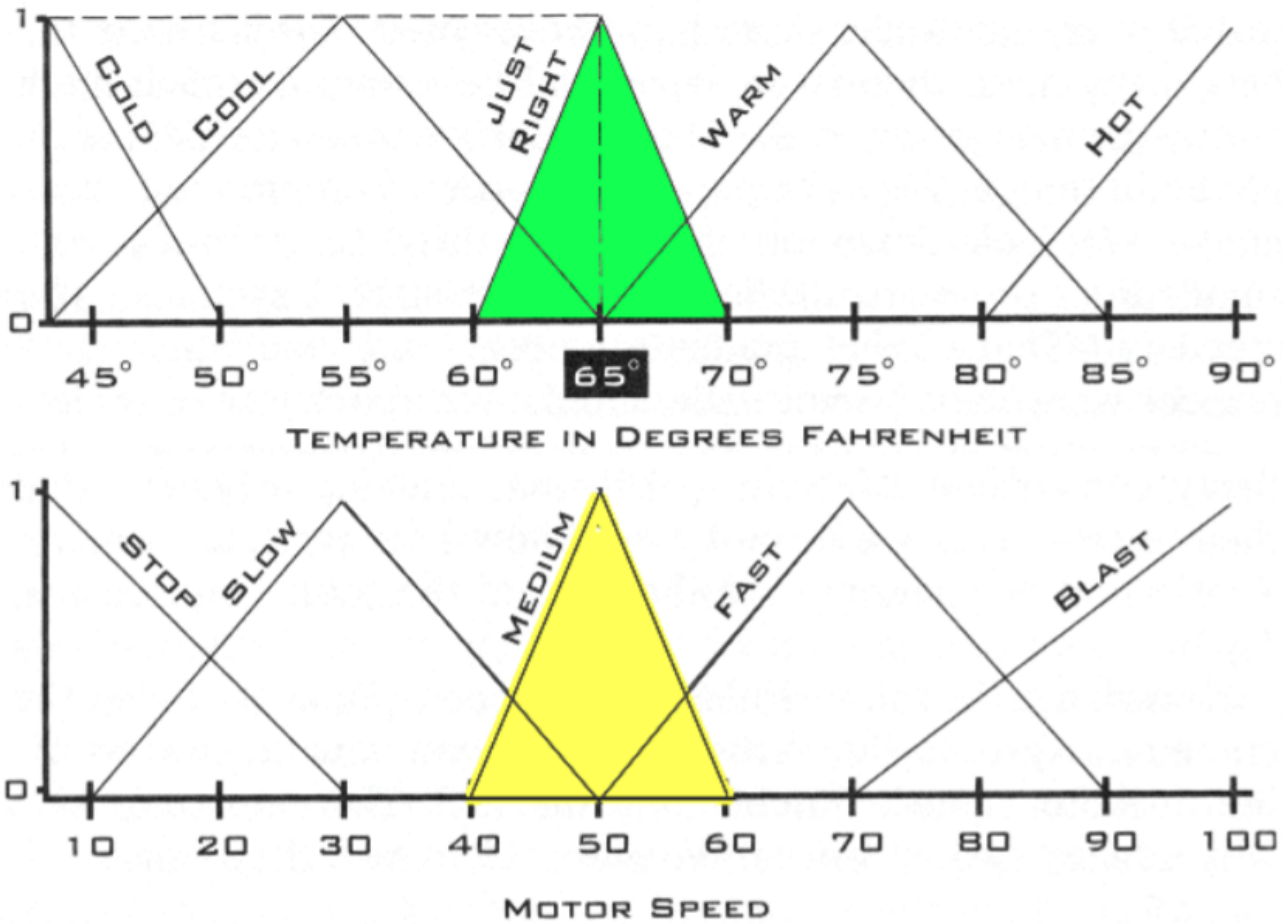
Several strategies exist for defuzzification

- The center of gravity (CoG) technique is very common

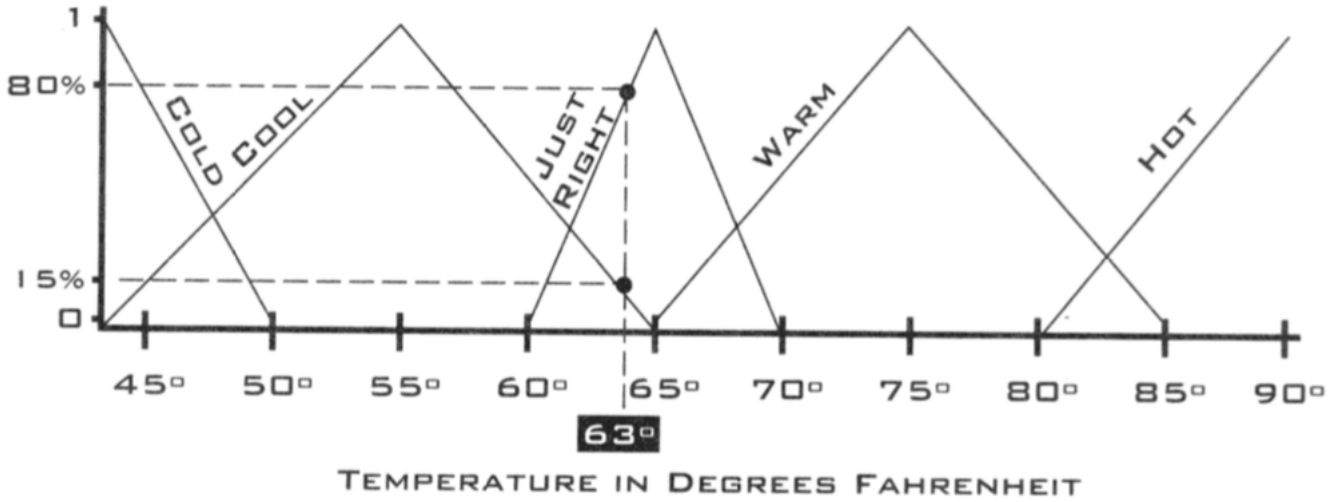
Other strategies include:

- Mean of maximum
  $\rightarrow$ The defuzzified result represents the mean value of all actions, whose membership functions reach the maximum
- Weighted average method
  $\rightarrow$ Formed by weighting each output by its respective maximum membership degree
- ...

Simple case

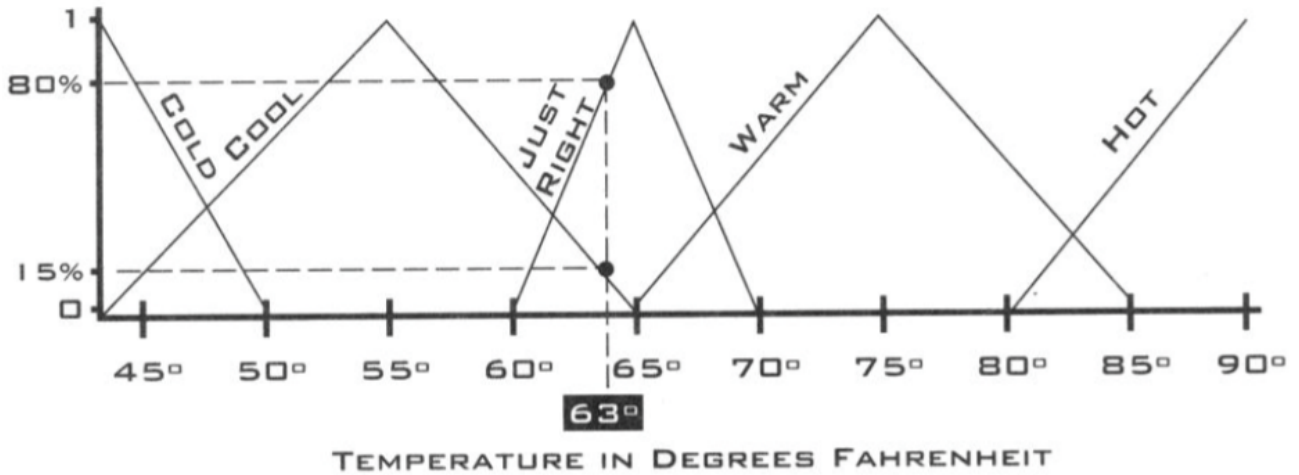

1
0
COLD
COOL
JUST RIGHT
WARM
HOT
45°
50°
55°
60°
65°
70°
75°
80°
85°
90°
TEMPERATURE IN DEGREES FAHRENHEIT
1
0
STOP
SLOW
MEDIUM
FAST
BLAST
10
20
30
40
50
60
70
80
90
100
MOTOR SPEED

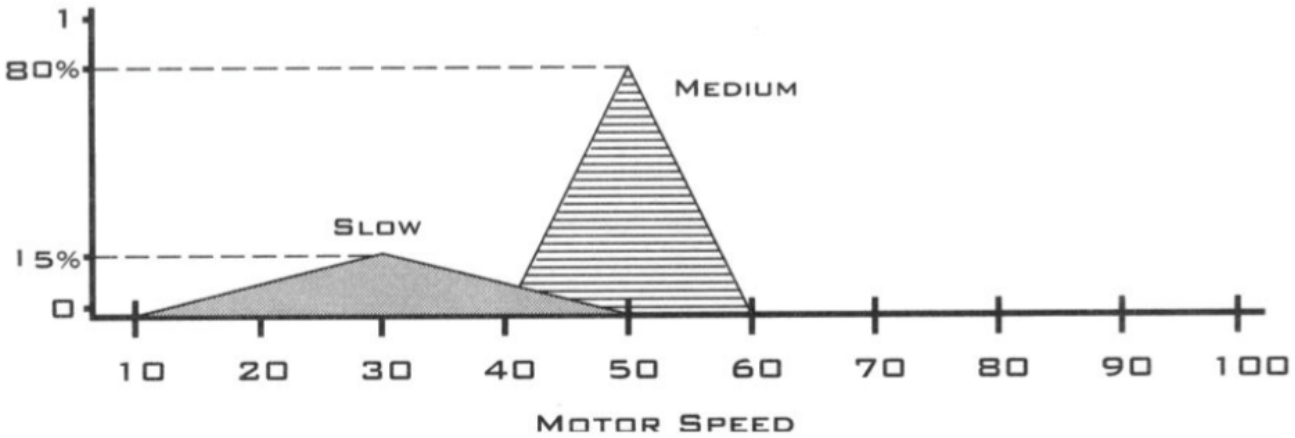
General case



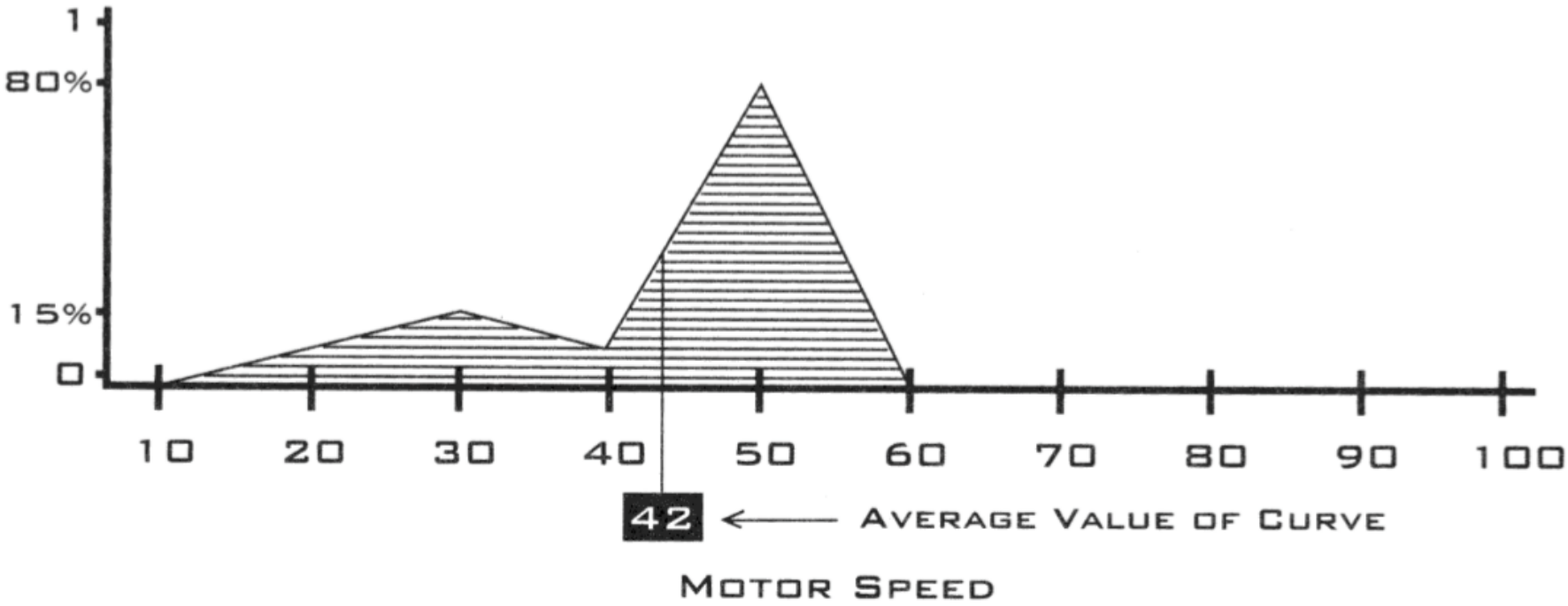
- IF temperature is **cool** THEN motor speed is **slow**
- IF temperature is **just right** THEN motor speed is **medium**

General case (cont.)





General case (cont.)



Curse of dimensionality

Fuzzy logic based control models (nonlinear modeling techniques) are affected by the *curse of dimensionality*

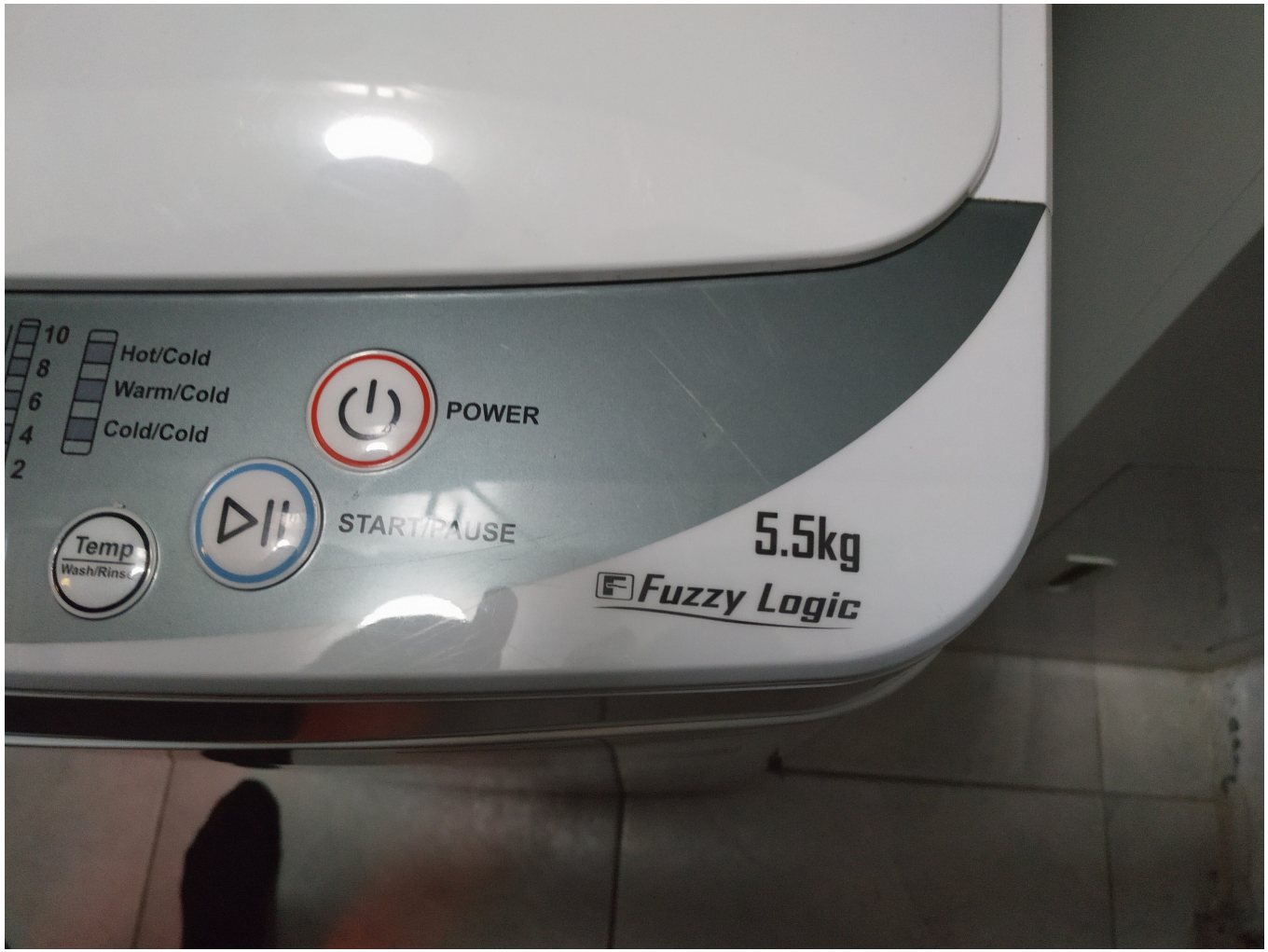
$\rightarrow$ If the number of inputs grows, the cost of both implementing the rule base and obtaining an output value increase exponentially.

Design Paradigms

- Hierarchical org.: Okay
- Reusable: Okay
- Modular: Bad
- Maintain.: Bad
- Human read.: Good
- Stateful: Bad
- Fast: Very good
  - The classic example of reactive approach
- Expressive: Bad
- Synthesis: Very good
- Understandable: Good
- Effort: Good

Fuzzy Control in Action

Fuzzy Control in Action



Summary

- Fuzzy logic provides a different way to approach a control problem
- It is based on natural language
- It allows to solve the problem without the need of a mathematical model
- FL based controllers require less implementation effort and are thus usually cheaper
- FL is inherently tolerant of imprecise data
- It can model nonlinear functions of arbitrary complexity
- Fuzzy logic can be combined with conventional control techniques

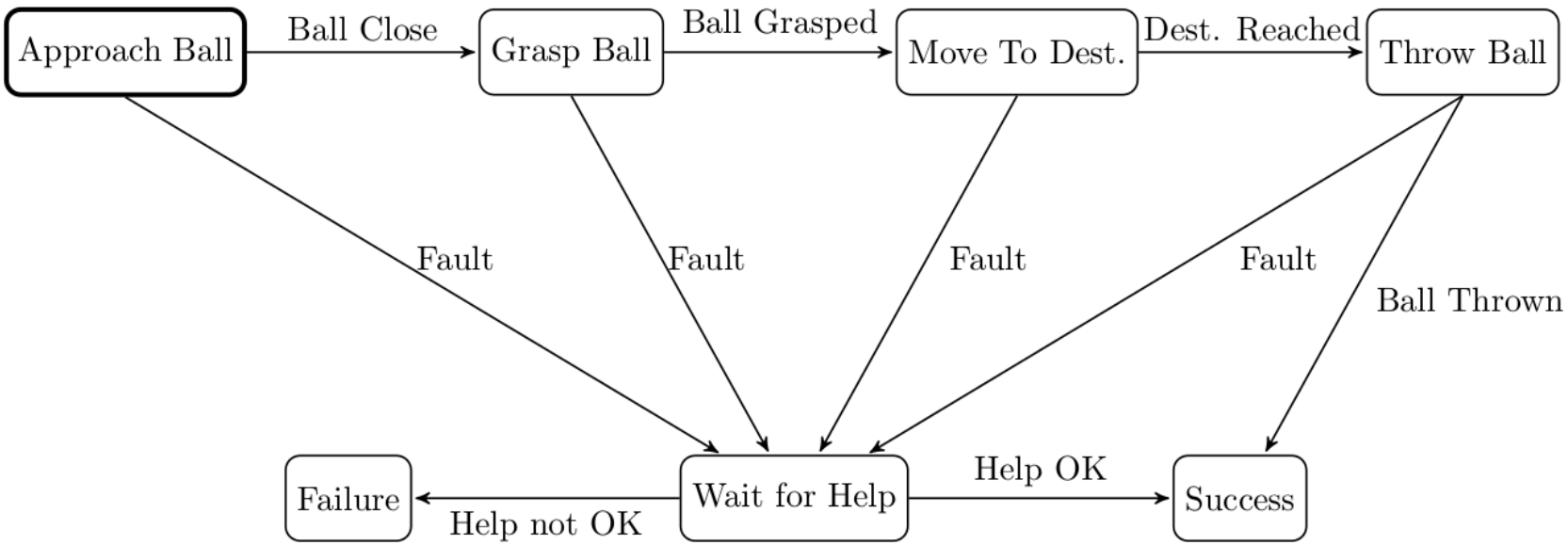
Summary (cont.)

- Fuzzy logic is not a cure-all!
- Fuzzy logic is a convenient way to map an input space to an output space
- However, many controllers can do a fine job without it
- Fuzzy logic *can* be a powerful tool for dealing with imprecision and nonlinearity

Finite State Machine

- Very common in computer science
- Often implicitly implemented
  - State is encoded in multiple variables of flags
- Good theoretical foundation
- Working principal
  - List of possible states
  - Transitions between those states
  - Start state
  - Check for transition conditions
  - Change state if condition is true
  - Act according to current state

FSM - Example



Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

FSM - Pseudo Code - Transition in States

```
class AbstractState:
    def run(self, blackboard):
        raise NotImplementedError
    def next(self, blackboard):
        raise NotImplementedError
class ApproachBall(AbstractState):
    def run(self, blackboard):
        # send some walking commands
    def next(self, blackboard):
        if blackboard.ball_distance < 1:
            return GraspBall()
        if blackboard.fault:
            return WaitForHelp()
        return self
class GraspBall(AbstractState): ...
class BallStateMachine:
    def __init__(self, blackboard):
        # the blackboard is some kind of object holding all information
        self.blackboard = blackboard
        self.current_state = ApproachBall()
        while true:
            self.run()
            sleep(0.1)
    def run(self):
        self.current_state.run()
        self.current_state = self.current_state.next()
```

FSM - Pseudo Code - Transition in Machine

```
class ApproachBall(AbstractState):
    def run(self, blackboard):
        # send some walking commands
class GraspBall(AbstractState):
    ...
class BallStateMachine:
    def __init__(self, blackboard):
        # the blackboard is some kind of object holding all information
        self.blackboard = blackboard
        fsm = StateMachine(inital=ApproachBall, states=[ApproachBall, GraspBall, ...])
        fsm.add_transition(from=ApproachBall, to=GraspBall, if=ball_close)
        fsm.add_transition ...
        while true:
            self.run()
            sleep(0.1)

    def ball_close(self):
        return blackboard.ball_distance < 1

    def run(self):
        self.fsm.get_current_state().run()
        self.fsm.check_transition()
```

FSM - Defining Transitions

- Transitions can be defined by the state (version 1)
- Or in the statemachine (version 2)
- This has pros and cons
- Pro in state
  - Decision can depend on "state of the current state"
    - Ex: "Wait5Sec" remembers time when state started
  - Simpler to implement
  - Easier to see to which state you go from one state
- Con in state
  - Danger of putting too much "state into a state", leading to an implicit HSM
  - Transitions are all distributed across states
  - More difficult to use if you have events
- Both versions can be found in libraries

FSM - Advantages and Disadvantages

Advantages:

- Commonly used in computer science
- Intuitive structure
- Ease of implementation

Disadvantages:

- Maintainability
- Scalability ("state explosion")
- Reusability
- No standardization

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

FSM - Conclusion

- Simple to understand and implement
- Very wide spread
- Use for small/trivial scenarios
- Stateful

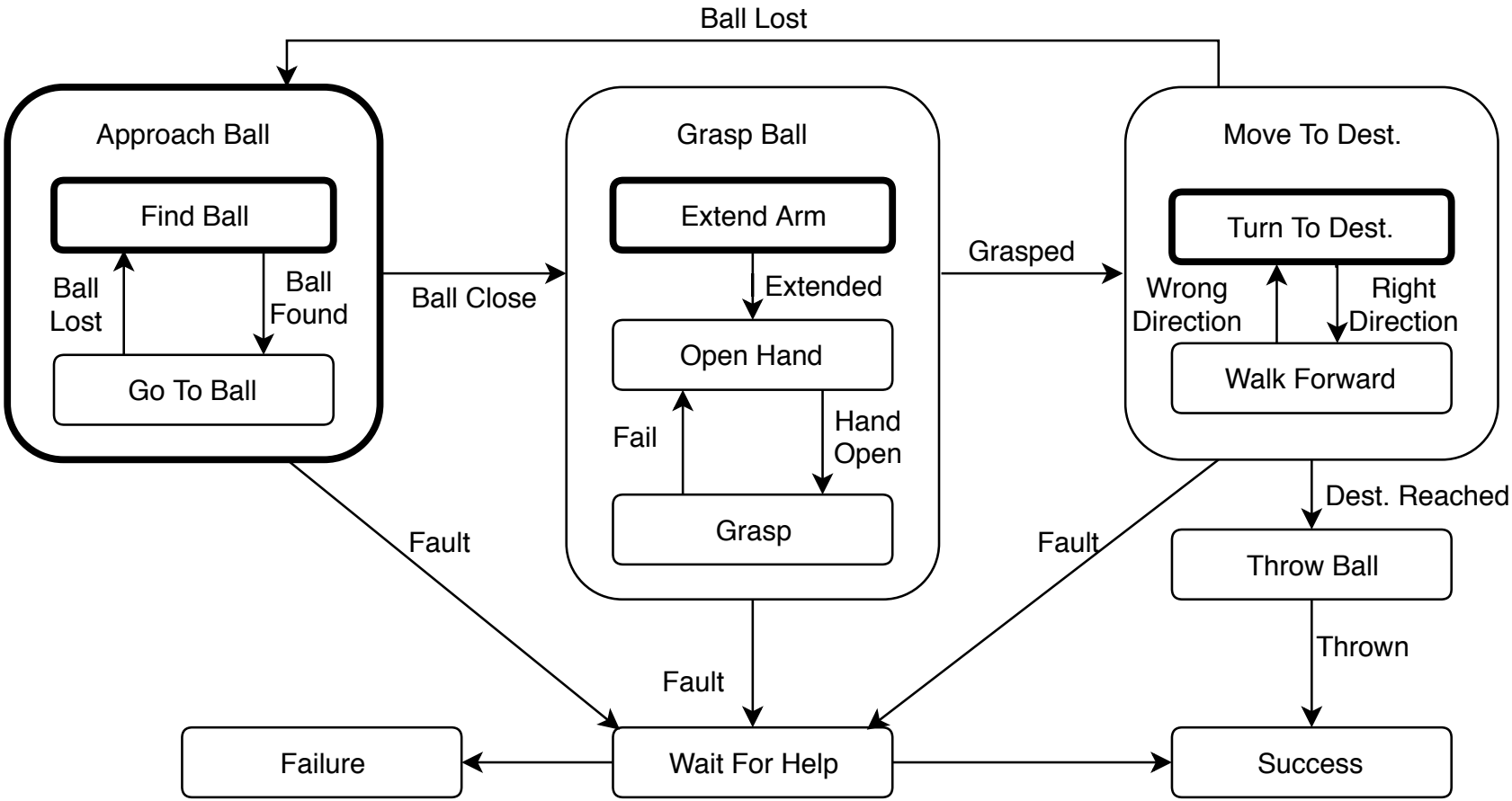
Libraries

- To many to list
- I recommend picking one which gives you graphical output for better debugging

Hierarchical State Machines

- Also known as State Charts (UML)
- Solve some of the problems of FSMs
- Introducing a hierarchical layout
- Each state can consist of substates
- States with substates are called superstates
- Generalized transitions connect superstates
- Each superstate has a start substate
- The number of overall transitions is reduced

HSM - Example


Ball Lost
Approach Ball
Find Ball
Ball Lost
Ball Found
Go To Ball
Ball Close
Grasp Ball
Extend Arm
Extended
Open Hand
Fail
Hand Open
Grasp
Grasped
Move To Dest.
Turn To Dest.
Wrong Direction
Right Direction
Walk Forward
Dest. Reached
Throw Ball
Thrown
Fault
Fault
Fault
Failure
Wait For Help
Success

HSM - Advantages and Disadvantages

Advantages:

- Modularity
- Behavior inheritance

Disadvantages:

- Maintainability
- Non intuitive hierarchy

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

HSM - Conclusion

- Still relative easy to implement
- Stateful
- Still comparably wide spread
- Useful in medium complex scenarios

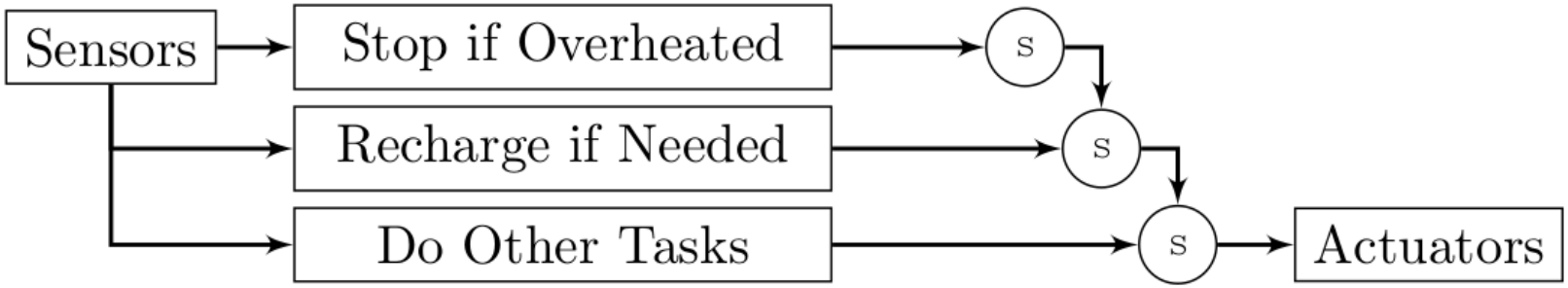
Libraries

- smach (ROS)
- pysm (Python)

Subsumption Architecture

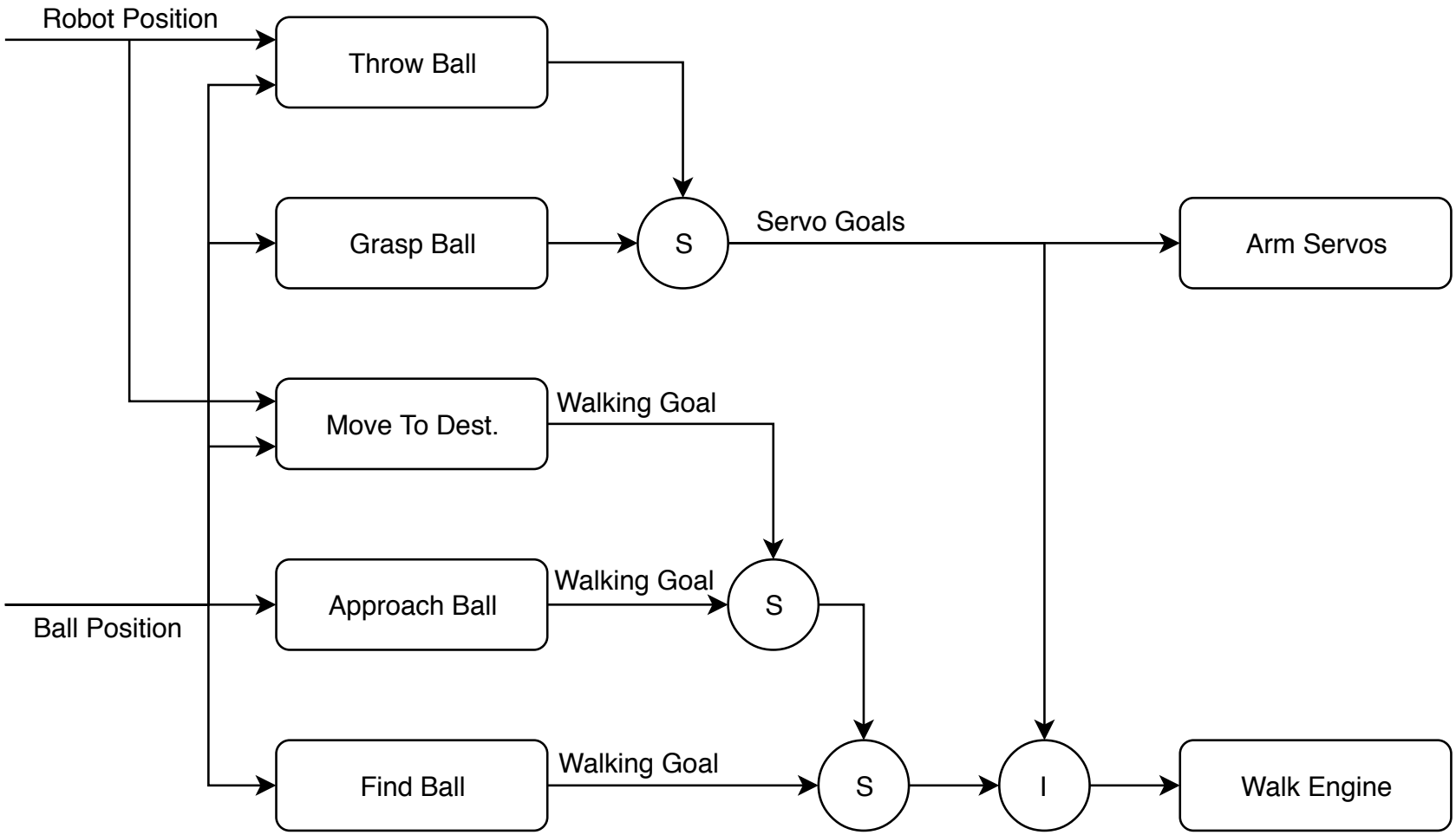
- Several modules
- Each implements one task
- All run in parallel
- Module are ordered by priority

Subsumption - Example



Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

Subsumption - Robot Example



Subsumption - Advantages and Disadvantages

Advantages:

- Modularity
- Hierarchy
- Reactivity

Disadvantages:

- Scalability
- Maintainability
- Not stateful

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

Subsumption - Conclusion

- Good for reactive systems
- Hard to handle time dimension
- Usable for small to medium complex systems
- Not widely used

Libraries

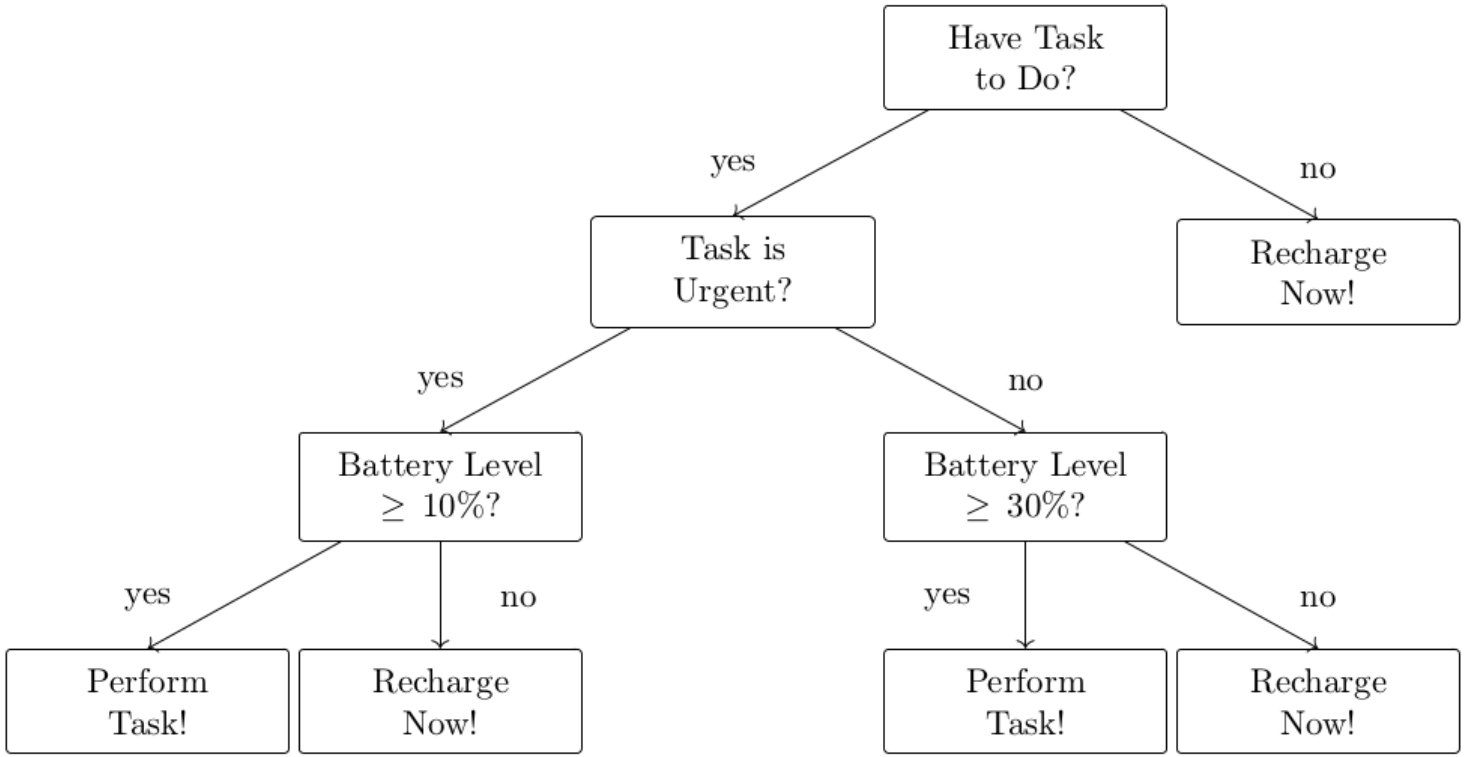
- subsuMeLib (C++) -outdated-

1. Decision Making

Decision Trees

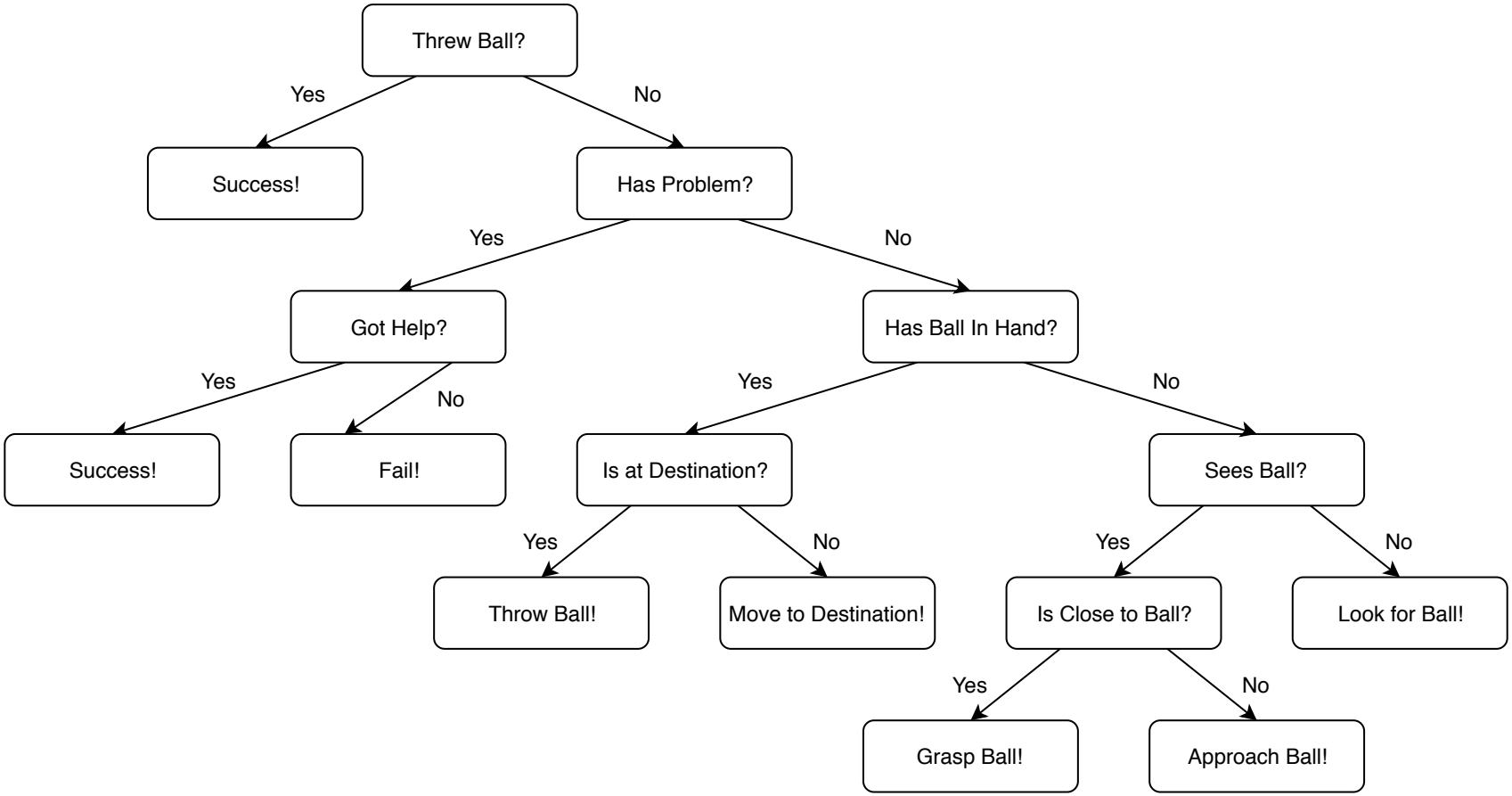
- Representation of nested if-then clauses
- Tree structure
- Internal nodes are predicates
- Leaf nodes are actions

DT - Example



Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

DT - Robot Example



DT - Advantages and Disadvantages

Advantages:

- Modularity
- Hierarchy
- Intuitive structure
- Clear division between actions and decisions

Disadvantages:

- Repetitions
- Maintainability
- Not stateful

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

DT - Conclusion

- Trivial to implement
- Using a framework rather than if-else can help with larger trees
- Widely (implicitly) used in computer science
- Easy to use with machine learning
- Good for domains which have no time dimension

Libraries

- scikit-learn + dtreeviz (Python)

Behavior Trees

- Tree of nodes
- Internal nodes are control flow nodes
  - Sequence
  - Fallback
  - Parallel
  - Memory
- Leaf nodes are execution nodes
  - Action
  - Condition
- Root node sends out *ticks* in fixed frequency to its children
- Only nodes that receive a tick are executed
- Children return *Running*, *Success* or *Failure*

BT - Sequence Node

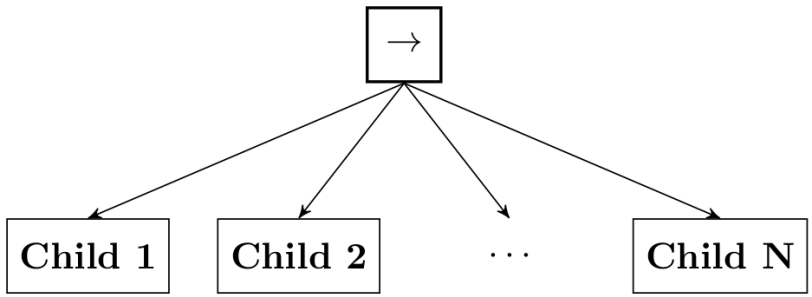


Fig. 1.2: Graphical representation of a Sequence node with N children.

Algorithm 1: Pseudocode of a Sequence node with N children

```
1 for i ← 1 to N do
2     childStatus ← Tick(child(i))
3     if childStatus = Running then
4         return Running
5     else if childStatus = Failure then
6         return Failure
7 return Success
```

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Fallback Node

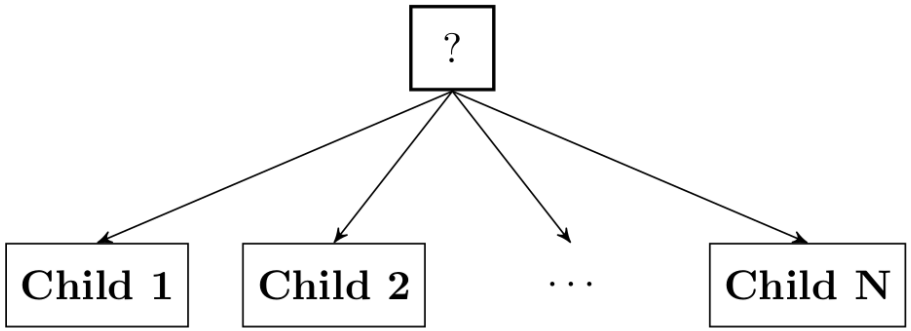


Fig. 1.3: Graphical representation of a Fallback node with N children.

Algorithm 2: Pseudocode of a Fallback node with N children

```
1 for i ← 1 to N do
2     childStatus ← Tick(child(i))
3     if childStatus = Running then
4         return Running
5     else if childStatus = Success then
6         return Success
7 return Failure
```

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Parallel Node

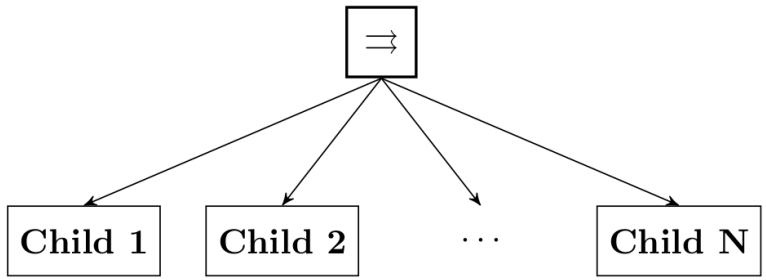


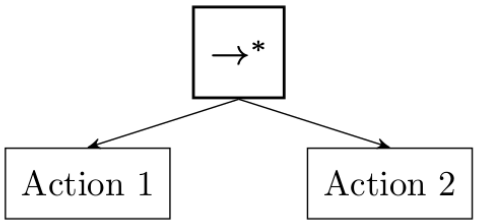
Fig. 1.4: Graphical representation of a Parallel node with N children.

Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

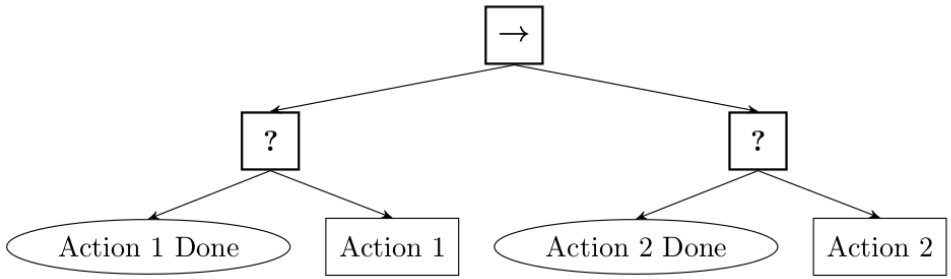
```
for i ← 1 to N do
    childStatus(i) ← Tick (child(i))
if Σ_{i:childStatus(i)=Success} 1 ≥ M then
    return Success
else if Σ_{i:childStatus(i)=Failure} 1 > N − M then
    return Failure
return Running
```

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Memory Node



(a) Sequence composition with memory.



(b) BT that emulates the execution of the Sequence composition with memory using nodes without memory.

Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

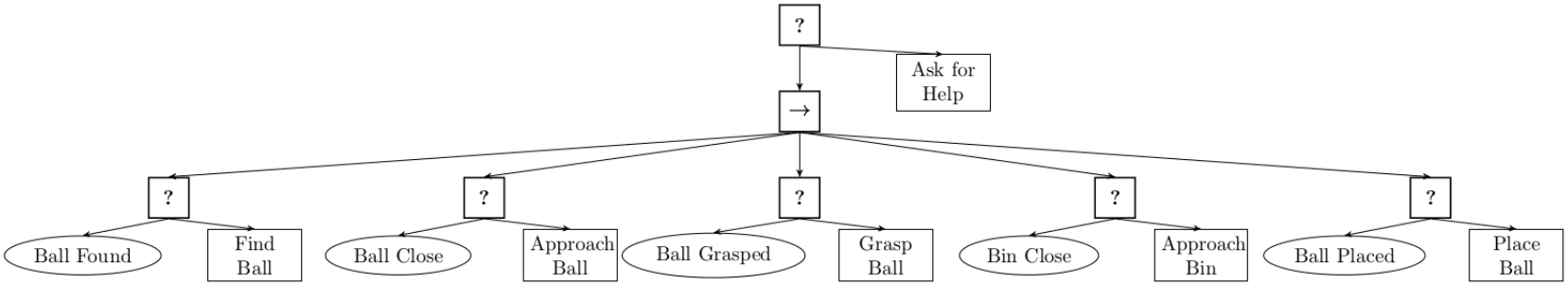
BT - Nodes

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	$\rightarrow$	If all children succeed	If one child fails	If one child returns Running
Parallel	$\rightrightarrows$	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

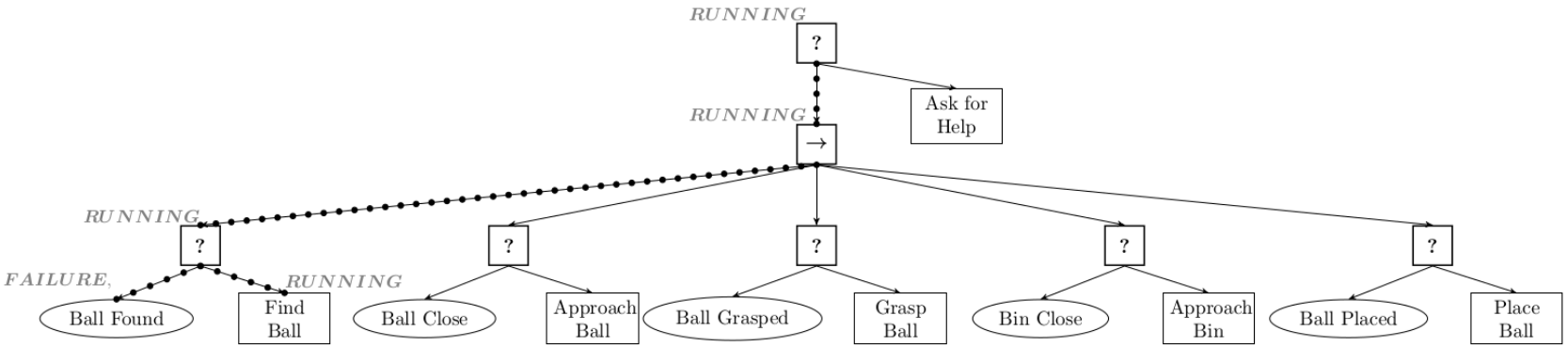
Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Example

Structure of the BT



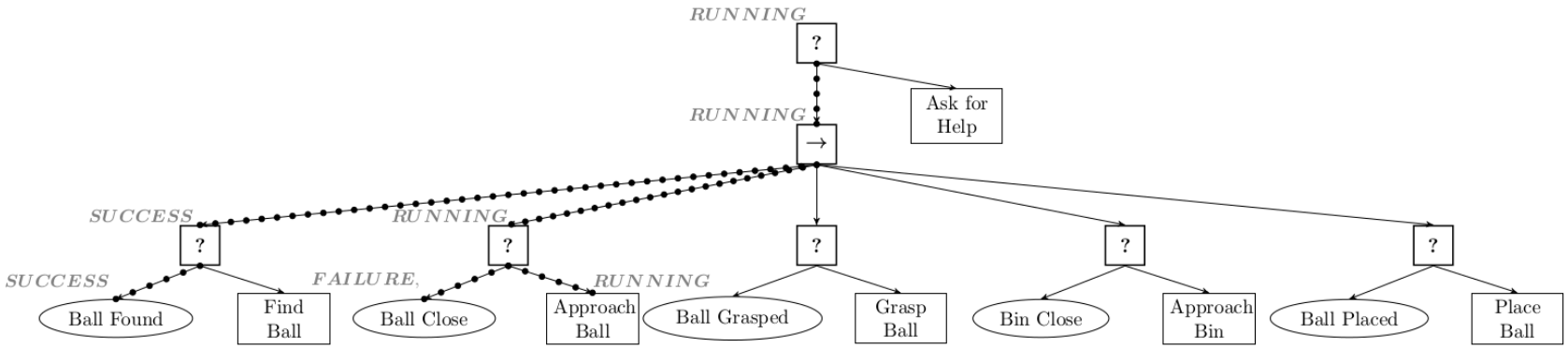
Start with searching the ball



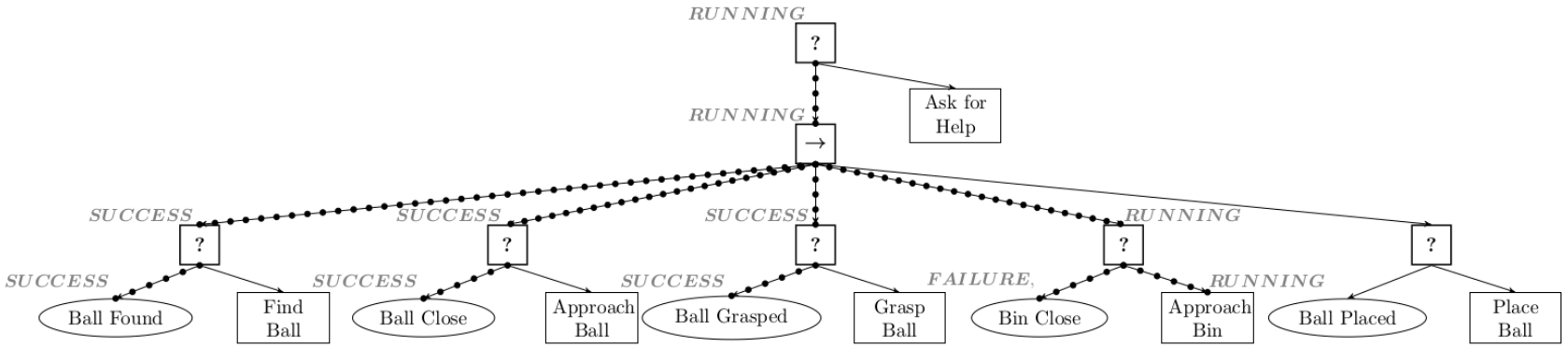
Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Example

Robot is approaching the ball

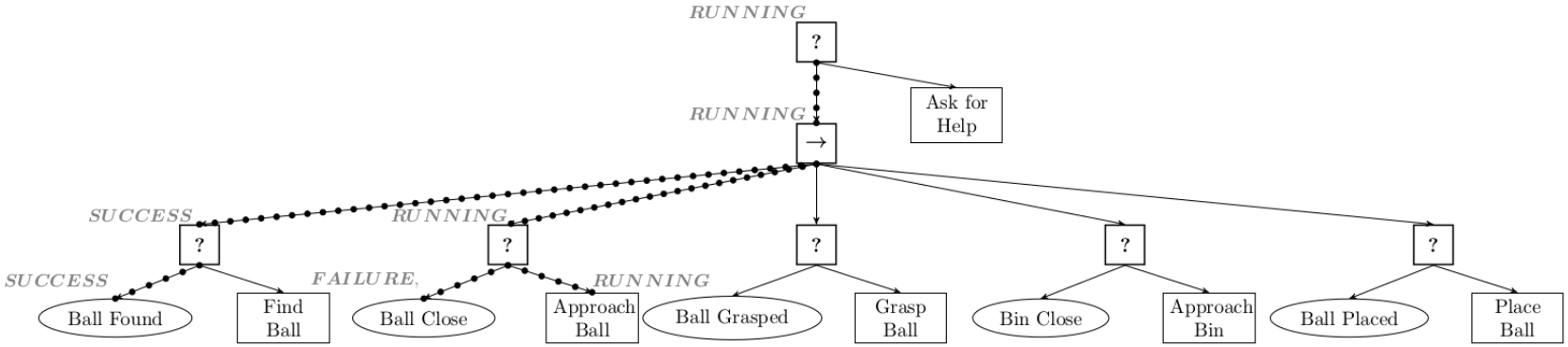


Robot is approaching the bin



Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI: an Introduction." 2017

BT - Example



Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

BT - Real World Example

Video

Advantages and Disadvantages

Advantages:

- Good modularization and code reuse
- Good maintainability
- Frameworks and knowledge from usage in game industry
- Well formalized elements
- Parallelism possible

Disadvantages:

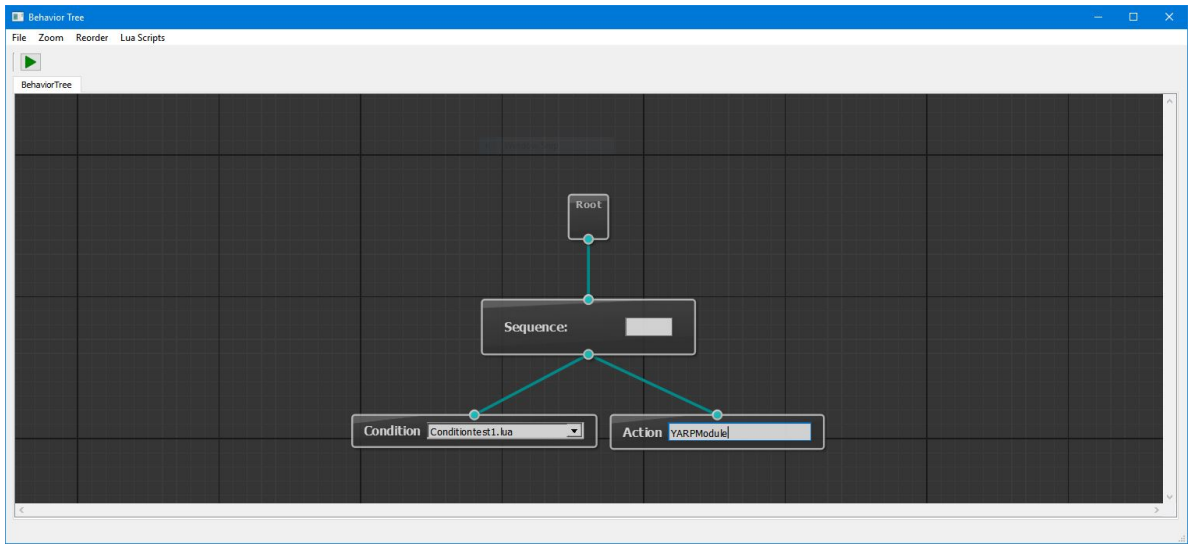
- Concept less intuitive
- Engine is complicated to implement
- Due to parallel activation current state difficult to see
- Testing preconditions indirect

BT - Conclusion

- Very powerful
- Good for large scenarios
- Also usable in smaller scenarios, but a bit overkill
- More complicated to learn but worth it

BT - Libraries and Tools

- Implemented in most big game engines
  - Unity
  - Unreal
  - ...
- behavior_tree (ROS package)
- YARP-Behavior-Trees (YARP Library)



Dynamic Stack Decider

Disclaimer: I was co-developer of the DSD.
Obviously it is good in my eyes.
Be critique about what I say.

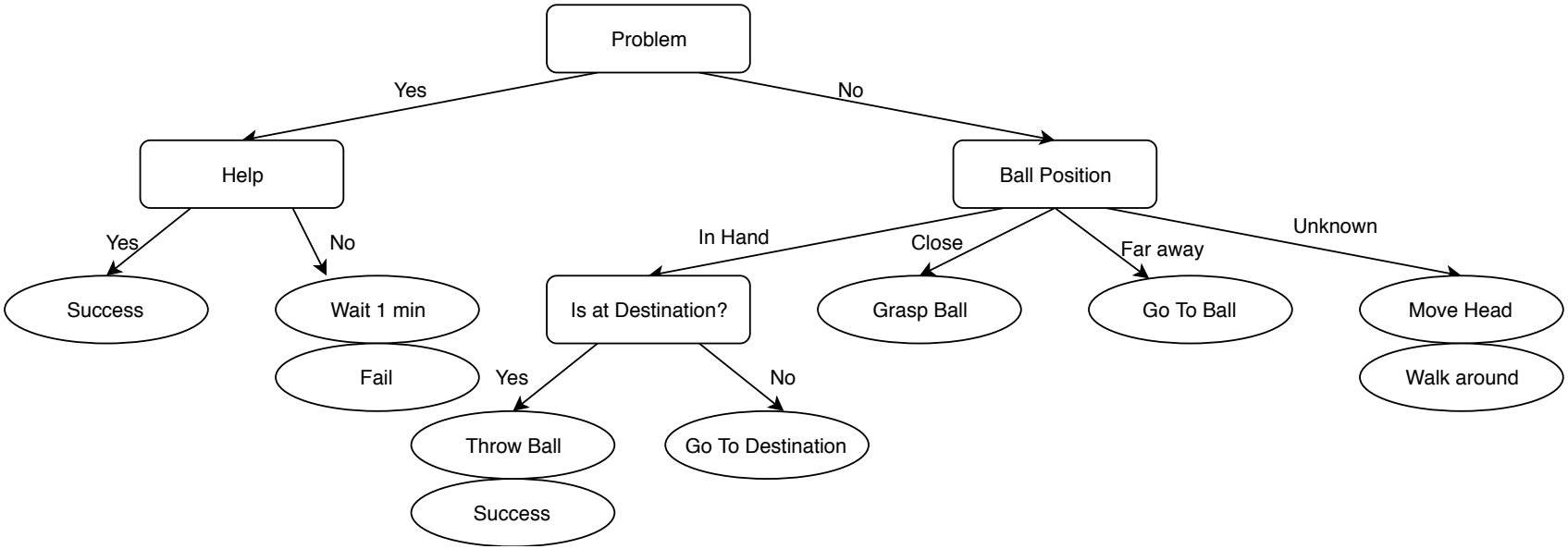
DSD - Motivation

- FSMs/HSMs not usable for non trivial problems
- BTs complicated to understand/implement, expensive to run
- Why not try to combine the advantages of the existing approaches
  - Tree structure (DT, BT)
  - Decision as internal nodes (DT)
  - Clear state (FSM, HSM)
  - Semantic transitions (FSM, HSM, DT)

Dynamic Stack Decider

- Tree like structure
  - Internal nodes are decisions (no time component)
  - Leaf nodes are actions (time component)
- Decisions provide a semantic outcome (string)
- Tree structure is defined by a simple domain specific language (DSL)
- State consists of current active action and previous decisions
- Decisions can be reevaluated to easily recheck preconditions
- But not all decisions have to be re-decided every time
- Action can be concatenated as sequences

DSD - Example


Problem
Yes
No
Help
Ball Position
Yes
No
In Hand
Close
Far away
Unknown
Success
Wait 1 min
Fail
Is at Destination?
Grasp Ball
Go To Ball
Move Head
Walk around
Yes
No
Throw Ball
Success
Go To Destination

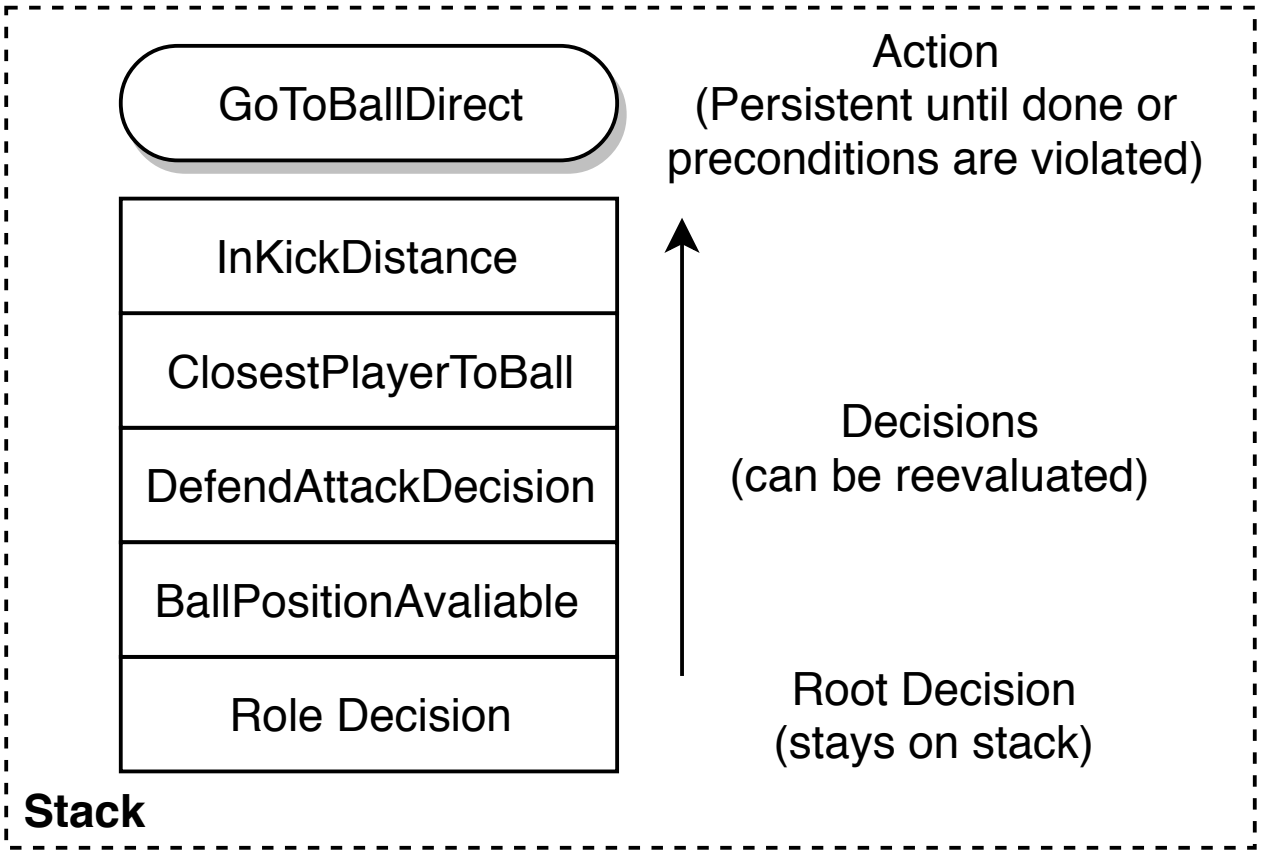
DSD - Domain Specific Language

```
-->BallBehavior
$Problem
    YES --> $Help
        YES --> @Success
        NO --> @Wait1Min, @Fail
    NO --> $BallPosition
        IN_HAND --> $IsAtDest
            YES --> @ThrowBall, @Success
            NO --> @GoToDest
        CLOSE --> @GaspBall
        FAR_AWAY --> @GoToBall
        UNKNOWN --> @MoveHead, @WalkAround
```

DSD - Code Example

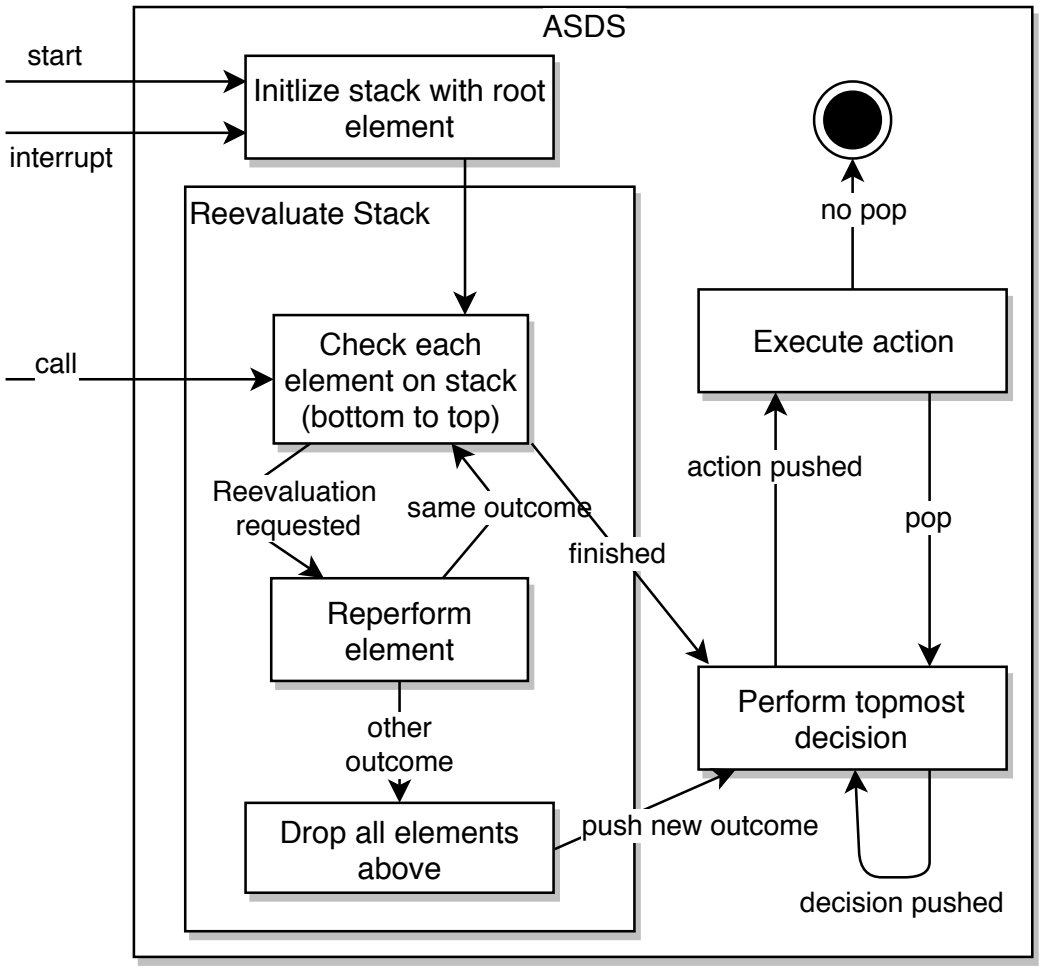
```
def BallPosition(AbstractDecisionElement):
    def perform(self):
        if not self.blackboard.ball_position:
            return "UNKNOWN"
        elif self.blackboard.ball_position < 0.5:
            return "IN_HAND"
        elif self.blackboard.ball_position < 1:
            return "CLOSE"
        else:
            return "FAR_AWAY"

    def get_reevaluate(self):
        return True

def GoToBall(AbstractActionElement):
    def perfom(self):
        # send some walking commands
def GrapsBall(AbstractActionElement):
    def __init__(self):
        # start grapsing animation

    def perform(self):
        if grasping_animation.is_finished():
            self.pop()
```

DSD - Stack



Poppinga, Martin and Bestmann, Marc. "DSD - Dynamic Stack Decider. A Lightweight Decision Making Framework for Robots and Software Agents" (2019) in peer-review process

DSD - Flow


ASDS
start
interrupt
Initlize stack with root element
Reevaluate Stack
call
Check each element on stack (bottom to top)
Reevaluation requested
same outcome
finished
Reperform element
other outcome
Drop all elements above
push new outcome
Perform topmost decision
decision pushed
action pushed
pop
Execute action
no pop

Video

Example video
https://www.youtube.com/watch?v=tZnBZsUAVqs (1:50)

Advantages and Disadvantages

Advantages:

- Clear seperation of decisions and actions
- Semantic transitittions
- Time component of actions
- Clear state and previous decisions
- Simple checking of preconditions
- Different reevaluation timescales possible in the same DSD

Disadvantages:

- No parallelism
- Concept not perfectly intuitive

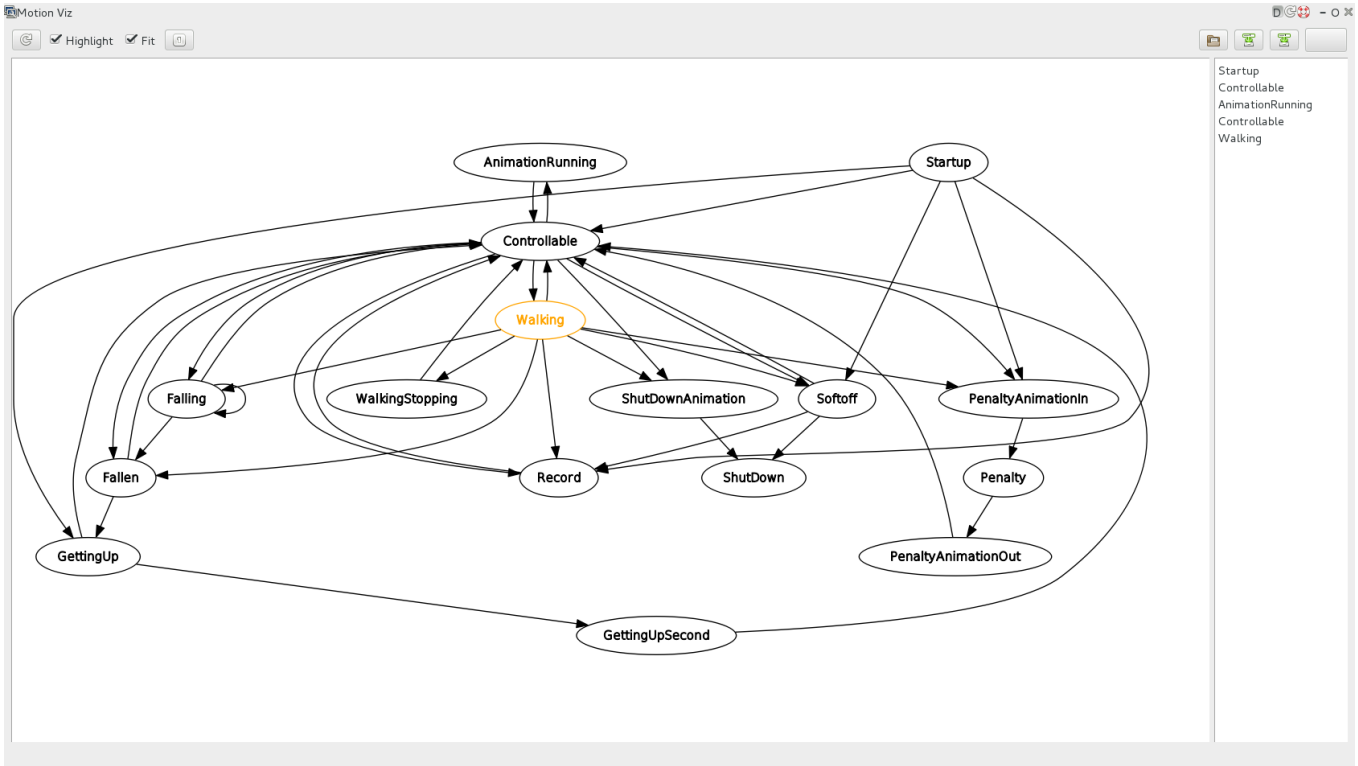
DSD - Conclusion

- Good for medium to complex scenarios
- Directly designed for robotics
- Not widely used

Libraries:

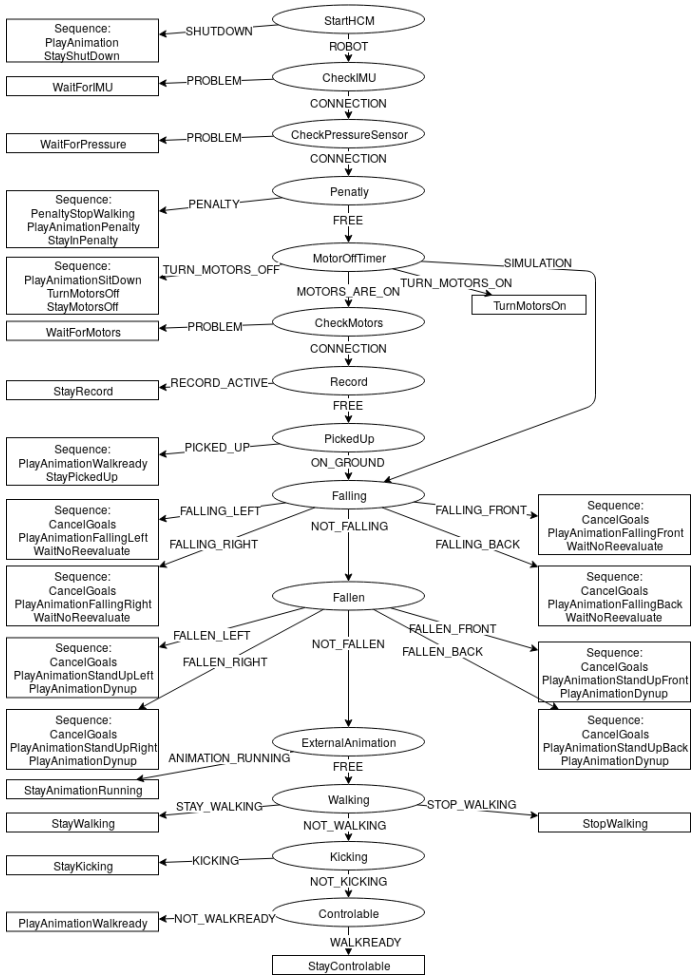
- dynamic_stack_decider (ROS)

Comparision FSM - DSD



The same control goal as FSM and DSD (with additional functions)

Comparision FSM - DSD



Summary

- There is not one perfect solution
- What fits depends highly on the use case
- Necessary to think which approach you want to use
- Besides pros and cons of the approaches there are other factors
  - What is currently used in the project
  - Where do you or your colleagues have already experience in
  - Is there a library available for your middleware

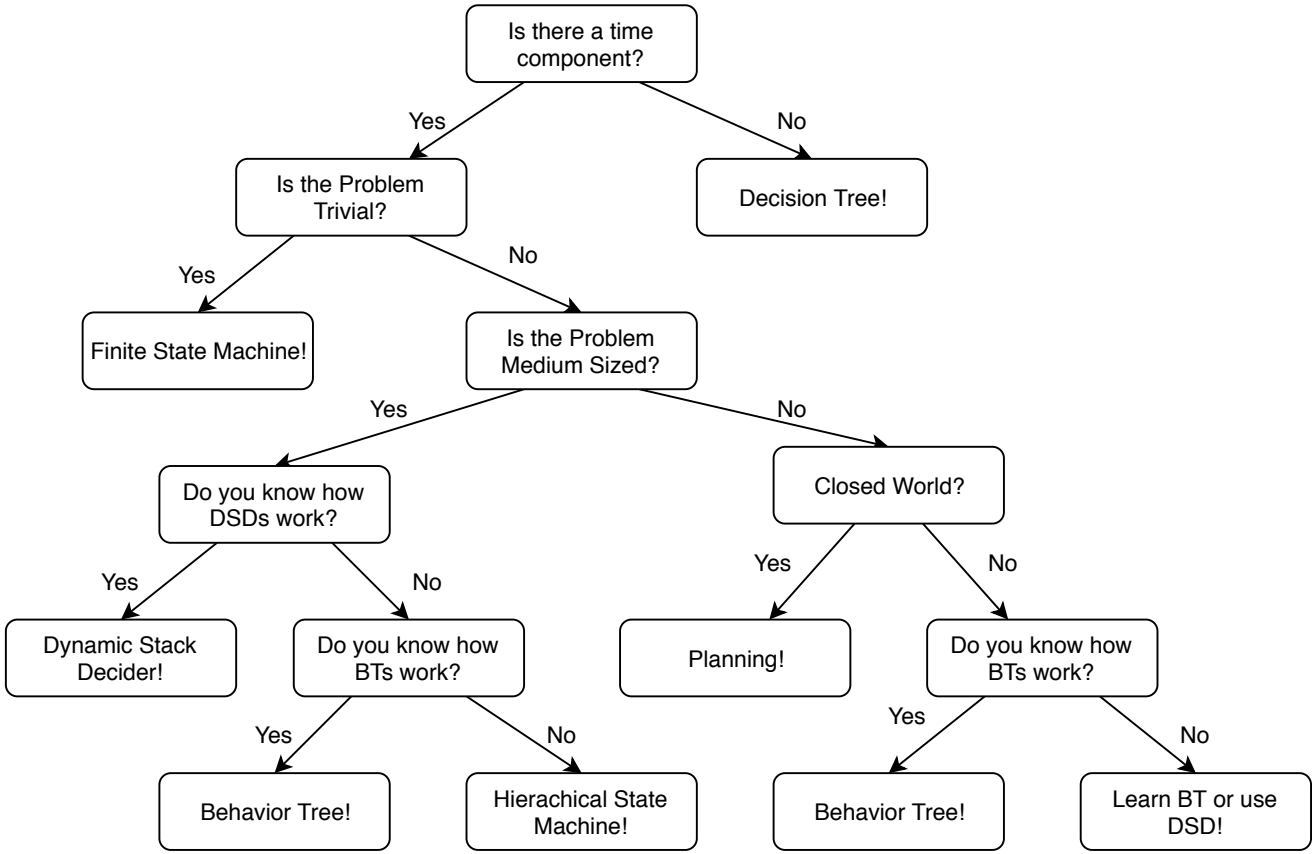
Comparison (subjective)

	SR	FL	FSM	HSM	Sub.	DT	BT	DSD
Hierarchical org.	-	○	-	+	++	+	+	+
Reusable code	+	○	-	○	-	+	+	++
Modular design	++	-	-	+	+	+	+	+
Maintainability	○	-	-	○	-	-	+	+
Human readable	+	+	-	○	-	+	○	++
Stateful	++	-	+	+	-	-	+	+
Fast	- -	++	-	-	+	+	○	+
Sufficiently expressive	+	-	+	+	-	+	+	+
Suitable for synthesis	+	++	+	+	-	+	○	+
Understandability	+	+	+	+	+	++	-	-
Implementation effort	-	+	++	+	-	++	-	○

(partly) Colledanchise, Michele, and Petter Ögren. "Behavior Trees in Robotics and AI, an Introduction.", 2017

Choosing an Architecture

My personal subjective proposal on choosing an architecture



Layered Architectures

- Complex systems often combine multiple approaches together
- Getting the best of both worlds
  - Reactive
  - Sophisticated planning
- One of the most used approaches to do this is using layers
- Upper most layer is deliberative, using some planning approach
- Only calls some *skills* (e.g. GoToPoint, PickUpBall)
- Those are implemented by state machines
- Each state may have an implicit decision tree to decide on the transition

Discuss Some Examples

Let's discuss which approach should be used for some examples