Behavior Representation

AD-HOC BEHAVIORS

FINITE STATE MACHINES

DECISION TREES

BEHAVIOR TREES

AUTONOMOUS AGENTS SYSTEMS 8/10/2025

Ad-hoc Behavior

Ad-hoc behavior is a behavior created or configured in a specific, punctual, and improvised way to solve a particular problem or situation, without following a general model or a structured approach.

can be a simple rule or routine created for an agent to react to a specific situation

"if battery below 10%, shut down immediately"

Ad-hoc Behavior

Definition: Fast, straightforward rules (if/else)
Advantages: Simple, fast
Cons: Poor scalability

Key features:

- Case specific, Fast and practical, Little generalization, Direct solution.
- Designed for a **specific situation** or environment, not necessarily reusable or scalable.
- Often developed to "get the **job done**" in the **short term**, without long-term planning.
- It hardly works well outside the context in which it was created.
- It is **usually a rule**, logic or behavior "done by hand" to meet an immediate requirement.

It can be useful for prototypes, testing, or situations where a complex solution isn't worth the effort. However, too many ad-hoc solutions can lead to systems that are difficult to maintain, with too many "patches" and no coherence.

Consider a robot that, when it encounters an obstacle, turns right to swerve.

This rule can be ad-hoc behavior if it is a quick fix, without considering other situations such as obstacles to the right or other possible routes.

```
using UnityEngine;
public class RobotAdhoc : MonoBehaviour
{
    public float battery = 100f;
    public Transform obstacle;
    void Update()
    {
 // Regra ad-hoc: se obstáculo perto, virar à direita
        if (Vector3.Distance(transform.position, obstacle.position) < 2f)
        {
            transform.Rotate(Vector3.up, 90f);
        }
  // Regra ad-hoc: se bateria baixa, parar
        if (battery < 10f)
        {
            Debug.Log("Desligar!");
            return;
        }
        transform.Translate(Vector3.forward * Time.deltaTime);
        battery -= Time.deltaTime;
    }
}
```

Ad-hoc Behavior

- Exercise 1

Add condition: if you find "Water" → stop

Finite State Machines

FSM, also called **finite-state automata**, are

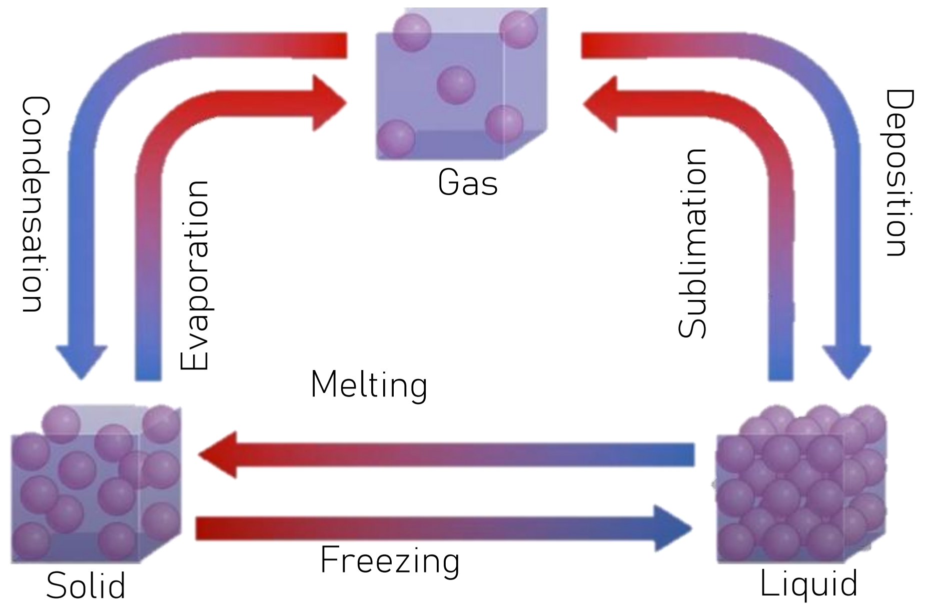
mathematical models used to represent agents that can be in a **finite number of states** and

that **transition** between those states based on **events** or **conditions**.

How it works:

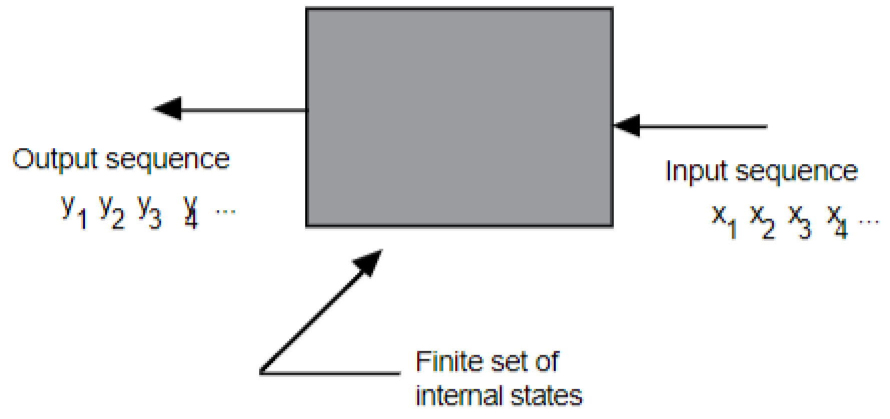
- Discreet number of states
- Can only exist one active state at a time
- Changes of state are triggered by transitions
- Optionally (outputs)

.



https://www.youtube.com/watch?v=I6LhmYTstd8

Finite State Machines


Output sequence
$y_1\ y_2\ y_3\ y_4 \ldots$
Input sequence
$x_1\ x_2\ x_3\ x_4 \ldots$
Finite set of internal states

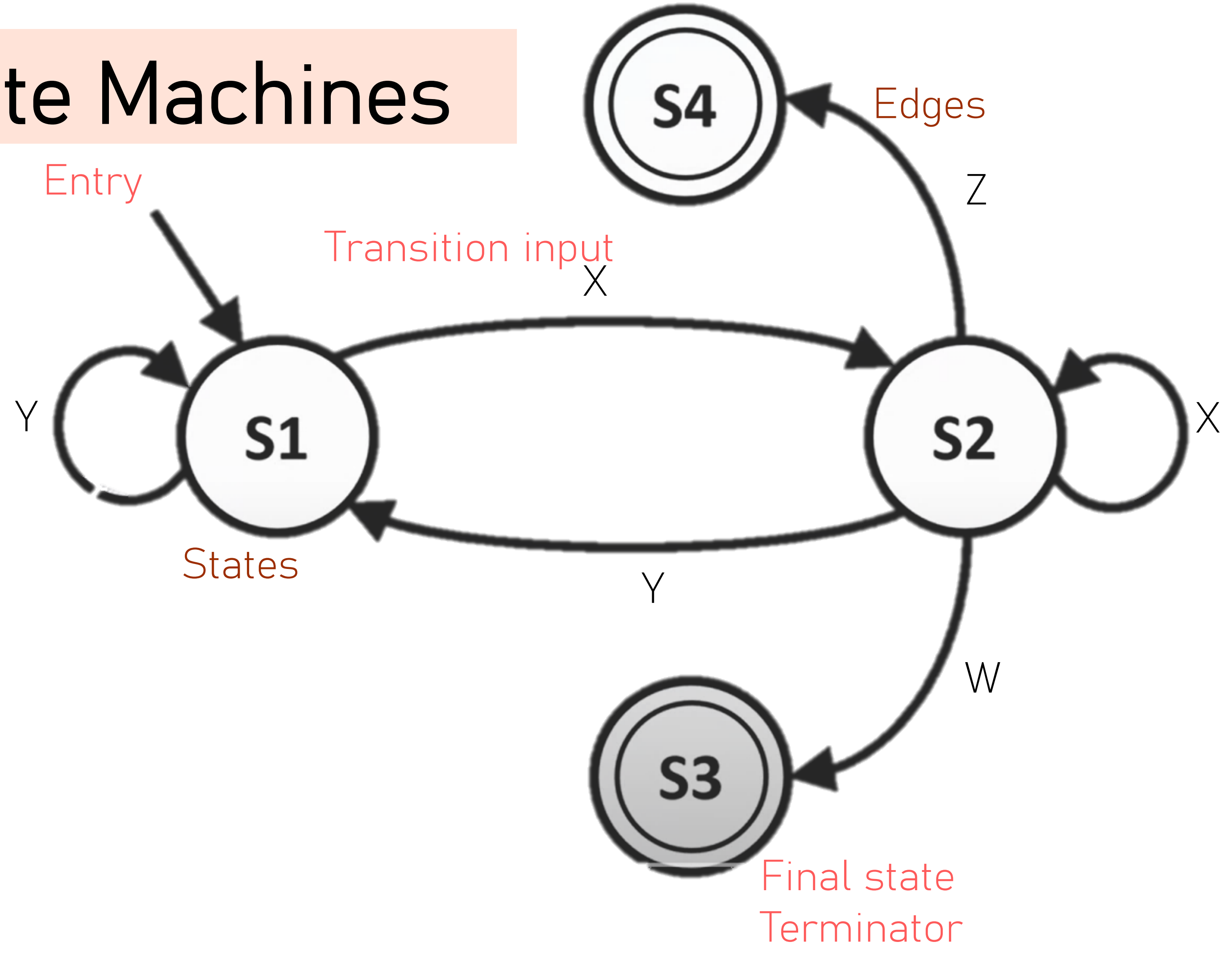
A reactive system whose **response** to a particular **stimulus** (a signal, or a piece of input) that is not the same on every occasion, **depending on its current "state"**.

The system changes state according to **predefined rules**

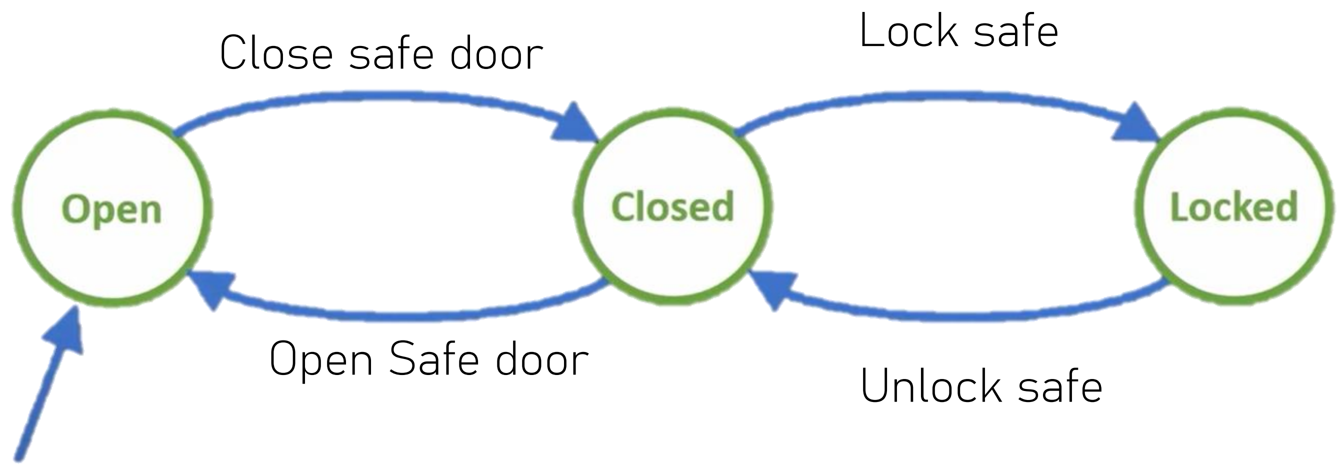
Each state corresponds to a **behavior** or **action**.

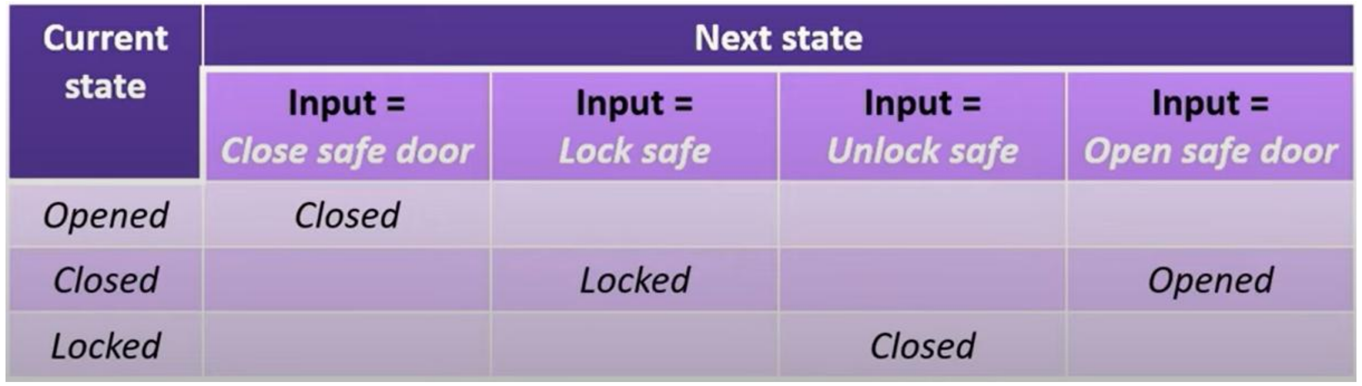
For example, in the case of a parking ticket machine, it will not print a ticket when you press the button unless you have already inserted some money.
Thus the response to the print button depends on the previous history of the use of the system.

Finite State Machines



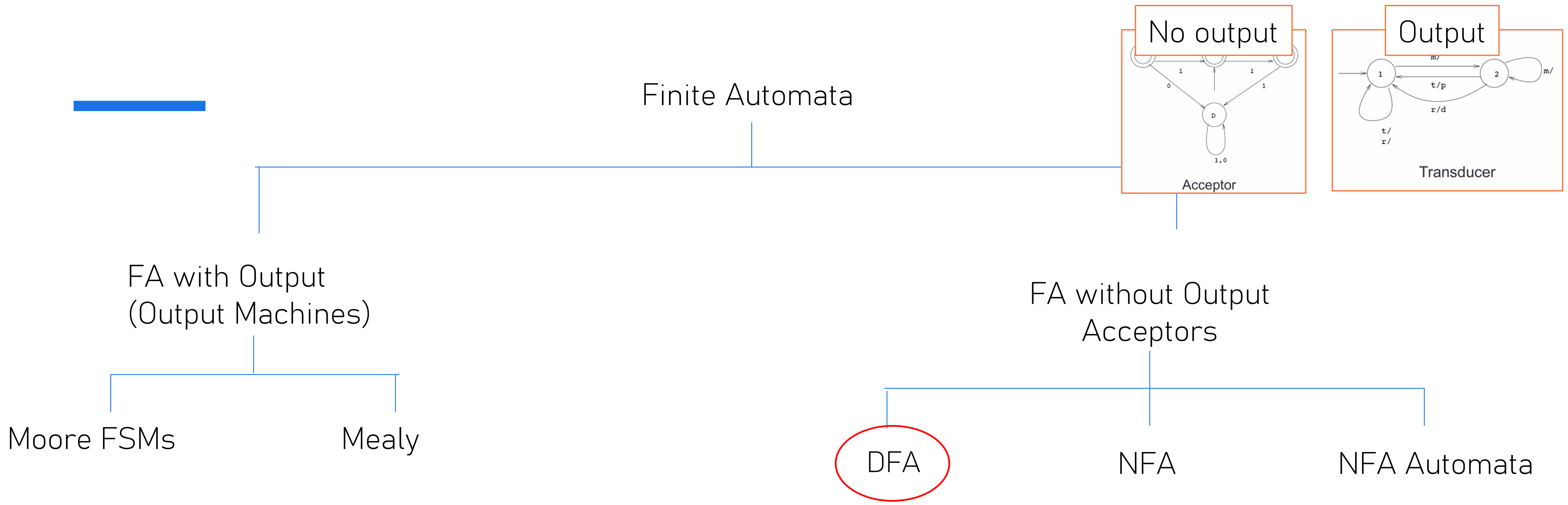
Finite State Machines





Current state	Next state			
	Input = *Close safe door*	Input = *Lock safe*	Input = *Unlock safe*	Input = *Open safe door*
Opened	*Closed*			
Closed		*Locked*		*Opened*
Locked			*Closed*	

State transition diagram



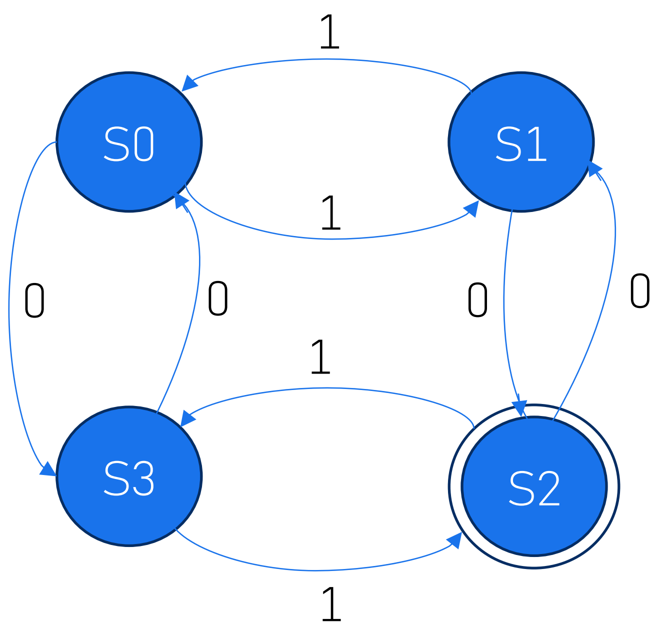
FSM are **deterministic** if, at any point during its operation,
there is exactly one possible next state given the current state and the current input symbol.
In other words, the machine's behavior is *completely determined* by:
- its **current state**, and
- the **current input symbol**.

There's **no ambiguity** or "choice" about what to do next.

Finite State Machines

A Finite State Machine is defined by (Σ,S,s0,δ,F), where:

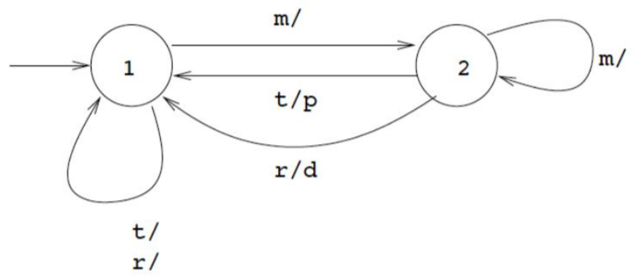
- S (or Q) is a finite, non-empty set of states.
- Σ is the input alphabet (a finite, non-empty set of symbols)
- s0 is an initial state, an element of S.
- δ is the state-transition function: δ : S x Σ → S
- F is the set of final states, a (possibly empty) subset of S.
- O is the set (possibly empty) of outputs



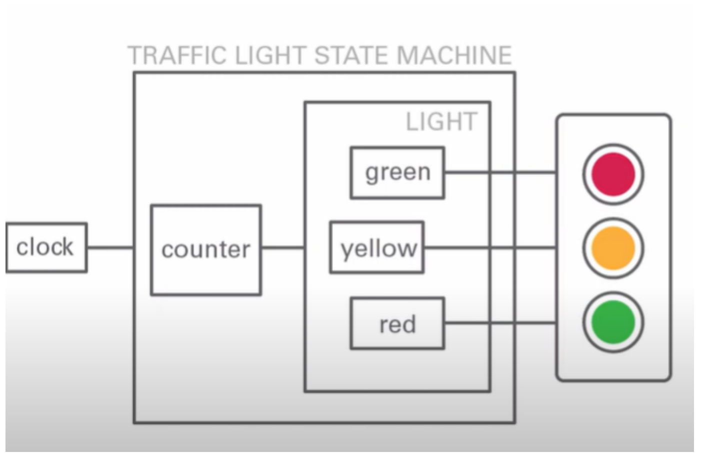
S = {S0, S1, S2, S3}
Σ = {0,1}
s0 = S0
Δ = S x Σ → S
F ={D}
O={S2}

	0	1
S0	S2	S1
S1	S3	S0
S2	S1	S2
S3	S0	S2

Finite State Machines



- S = {1, 2} : unpaid, paid
- Σ = {m, t, r} : inserting money, requesting ticket, requesting refund
- s0 = {1} : an initial state, an element of S.
- δ : transition function: δ : S x Σ →S
- F : Ø empty
- O = {p,d} : print ticket, deliver refund



- S={Red, Green, Yellow}
- Σ= {timer_tick]
- S0= Red
- δ(Green,green_timeout) = Yellow
- δ(Yellow,yellow_timeout) = Red
- δ(Red,red_timeout) = Green
- F=Ø (the empty set, because there is no “accepting” or “final” state)

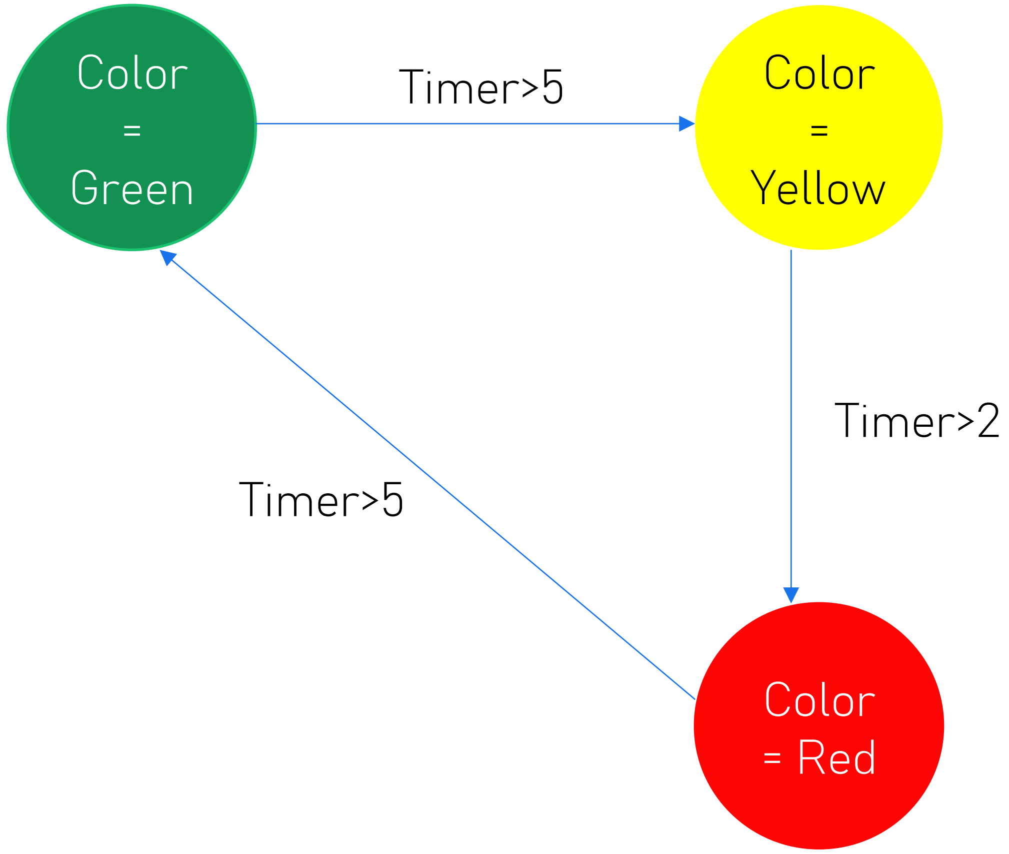
```
using UnityEngine;
public class TrafficLightFSM : MonoBehaviour
{
    enum State { Verde, Amarelo, Vermelho }
    State current = State.Verde;
    float timer = 0f;

    void Update()
    {
        timer += Time.deltaTime;
        switch (current)
        {
            case State.Verde:
                if (timer > 5f) ChangeState(State.Amarelo);
                break;

            case State.Amarelo:
                if (timer > 2f) ChangeState(State.Vermelho);
                break;

            case State.Vermelho:
                if (timer > 5f) ChangeState(State.Verde);
                break;
        }
    }

    void ChangeState(State newState)
    {
        current = newState;
        timer = 0f;
        Debug.Log("Novo estado: " + current);
    }
}
```



Finite State Machines

- Unity examples:
  - Traffic Light (TrafficLightFSM)
  - Automatic Door (DoorFSM)

- Exercice 2
- Modify the door to open slower or faster:
- Detect when player is nearby, and if so → keep it open

Finite State Machines

NPC agent
in game:

States (Each state represents a mode of behavior of the agent):	Idle (Stopped) Patrolling Chasing Attacking
Transitions (Rules that define when the agent switches state, based on events or conditions in the environment):	Idle → Patrolling (elapsed time) Patrolling → Chasing (detects enemy) Chasing → Attacking (gets close to the enemy) Attacking → Idle
Events/Conditions:	Sensors, signals, or variables that indicate the change in state

Finite State Machines

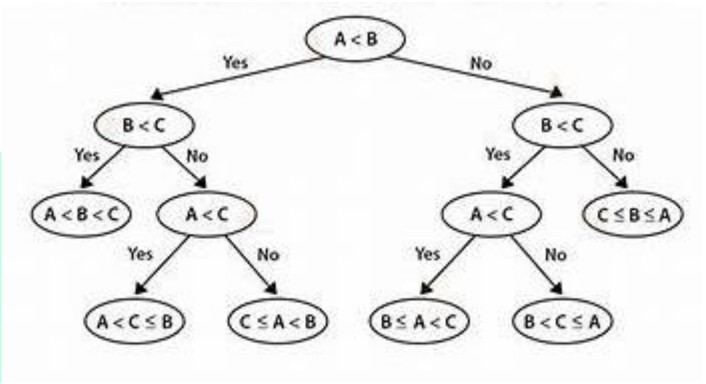
Strategies to mitigate

- Divide FSM into hierarchies (hierarchical state machines) for greater modularity.
- Combine FSM with other models (for example, use FSM for high-level and local behavior for low-level).

- Pros:
  - Definition: Clarity, easy to view, well-defined states + transitions
  - Advantages:
    - Allows for quick and explicit reactive behavior.
    - Easy deployment in controlled environments.
    - Good for tasks with few states and clear rules.

- Cons:
  - Explodes in complexity with many states
    - Can become very complex and difficult to maintain: An agent who needs to deal with multiple objectives and conditions (e.g., explore, avoid enemies, communicate) may need hundreds of states.
    - Little flexibility to adapt new behaviors without rewriting the entire FSM: adding or modifying behaviors implies tinkering with the entire machine.
    - It is difficult to deal with combinations of multiple factors (e.g., hunger + fear + aggression).
    - Lack of memory or history: FSM are good for present states, but they don't easily record past sequences.

Decision Trees



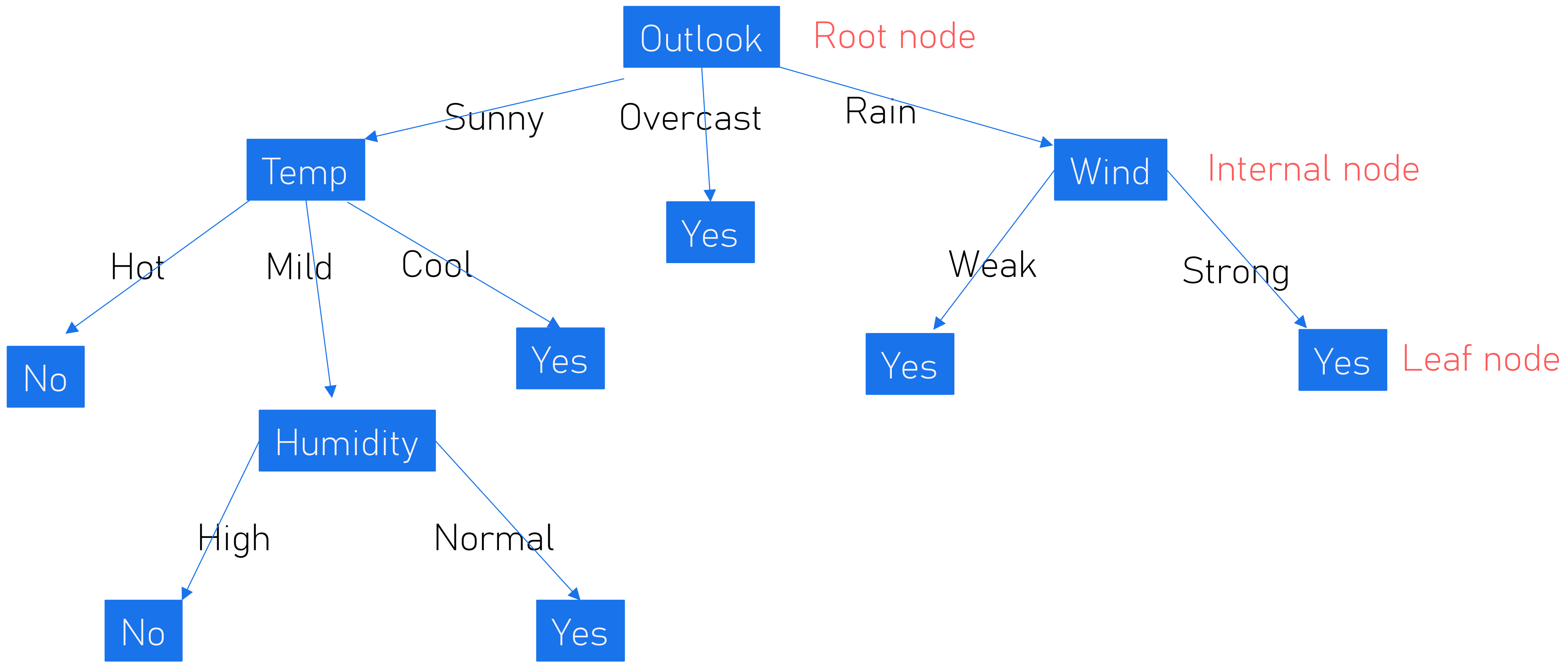
Decision trees are hierarchical structures where each **node** represents a **condition** or **test**, and each **leaf** represents an **action** or **decision**.

How it works:
The agent evaluates conditions in sequence, working down the tree until it reaches an action it should take.

Basic structure:
- Internal nodes: Logical tests (e.g., "enemy is visible?", "energy < 30%?").
- Leaves: Decisions or actions to be taken (e.g., "attack," "flee," "look for food").

Decision nodes – typically represented by squares
Chance nodes – typically represented by circles
End nodes – typically represented by triangles

Tenis game?



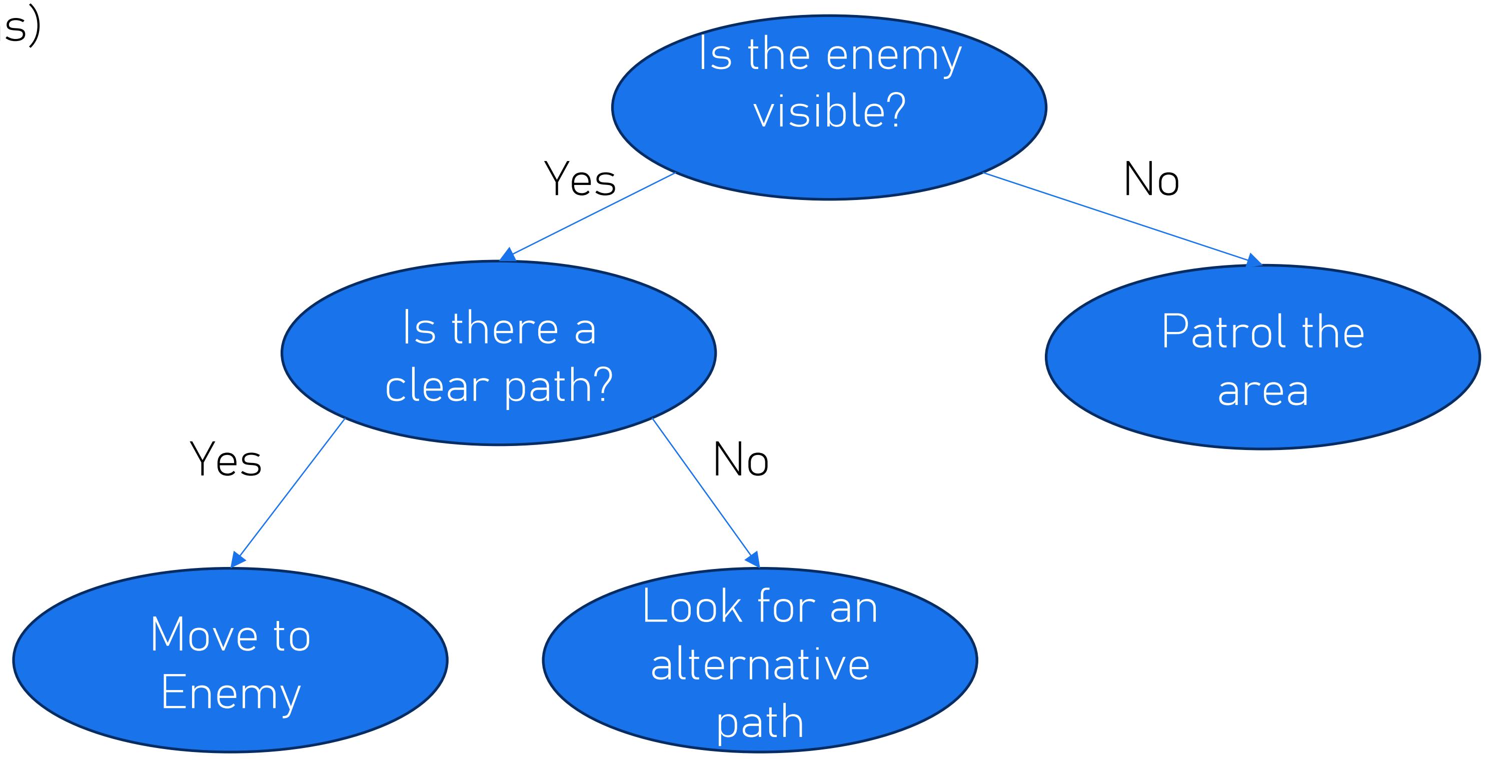
https://www.cs.toronto.edu/~axgao/cs486686_f21/lecture_notes/Lecture_07_on_Decision_Trees.pdf

Decision Trees

Decision tree definition (conditions → actions)

Advantages:
Clear hierarchy of conditions

Disadvantages:
Large trees can get difficult to manage



Rescue agent

Decision Trees

Application in autonomous agents

- Used for quick decision-making based on multiple conditions.

Example: a drone that decides whether to "fly over", "land" or "return to base" based on parameters (fuel, weather conditions, mission).

Consider an agent that
Attacks the enemy whenever it is nearby, in normal conditions patrols, and requires recharging when battery is low.

```
using UnityEngine;
public class SimpleDecisionTree : MonoBehaviour
{
    public float battery = 100f;
    public Transform enemy;
    void Update()
    {
        if (battery < 20f)
        {
            Recharge();
        }
        else if (Vector3.Distance(transform.position, enemy.position) < 3f)
        {
            Attack();
        }
        else
        {
            Patrol();
        }
    }
   void Recharge() => Debug.Log("Recharge...");
    void Attack() => Debug.Log("Attack!");
    void Patrol() => Debug.Log("Patrol...");
```

Decision Trees

Exercise 3.1

Create Decision Tree for Driverless Car:

- If critical battery → stop.
- If car near→ slow.
- If velocity < 10 → accelerate
- otherwise → cruise.

Exercise 3.2

- Add condition: if water > 30 and near→ divert

Decision Trees

Strategies

- Use decision trees to choose sub-behaviors within a larger controller.
- Integrate machine learning to generate trees that evolve as the agent gains experience.

Benefits

•**Modularity:** Sub-behaviors can be developed and tested independently.
•**Reactivity:** High-priority behaviors can interrupt others when needed.
•**Scalability:** Easy to add new behaviors.
•**Debugging & Maintenance:** Hierarchical structure helps locate issues.
•**Conditional Decisions:** Handles multiple criteria effectively.
•**Transparency:** Clear and modifiable decision logic.
•**Automation Potential:** Can be built using machine learning (e.g., Decision Tree Learning).

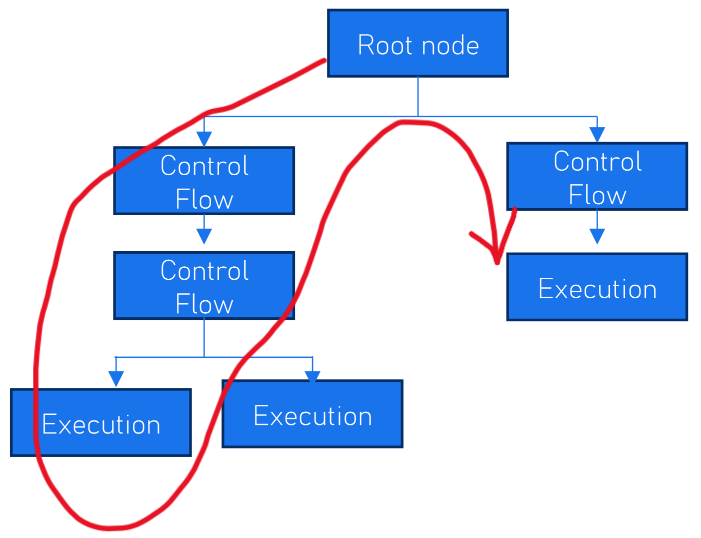
Technical Limitations

- Large trees are **hard to maintain** and interpret.
- Deep trees can **reduce performance**.
- Poor at representing complex or sequential behaviors.
- Limited support for internal states (memory).
- Mostly suitable for **one-off, sequential decisions**.

Behavior Trees

BT are hierarchical models for controlling the behavior of agents

widely used in games and robotics.

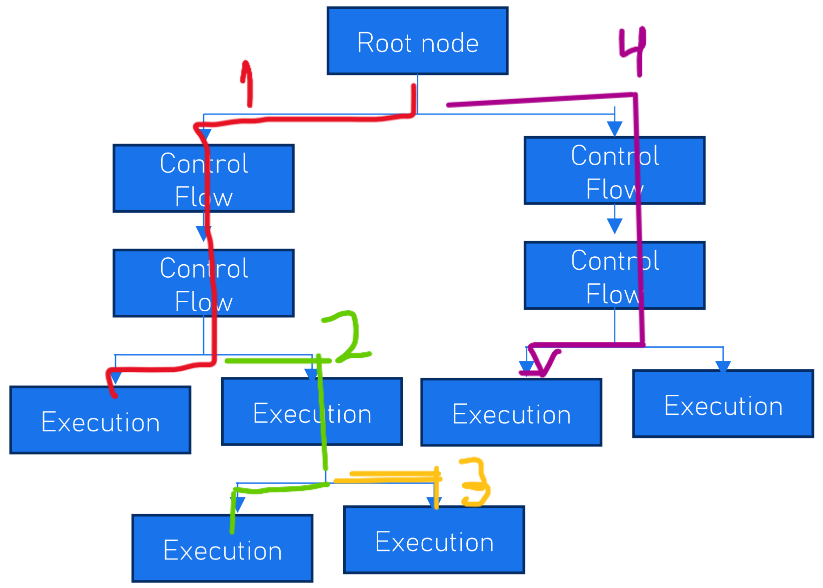


How it works:
Behavior trees combine **actions**, **conditions**, and **controllers** (such as sequences, selectors) to create reactive and modular behaviors.
BT uses a signal called **Tick** started from the Root Node and **propagated to its children** (left-to-right) until **certain conditions happen**.
A node has a program that is **executed** only when it **receives a Tick** signal.

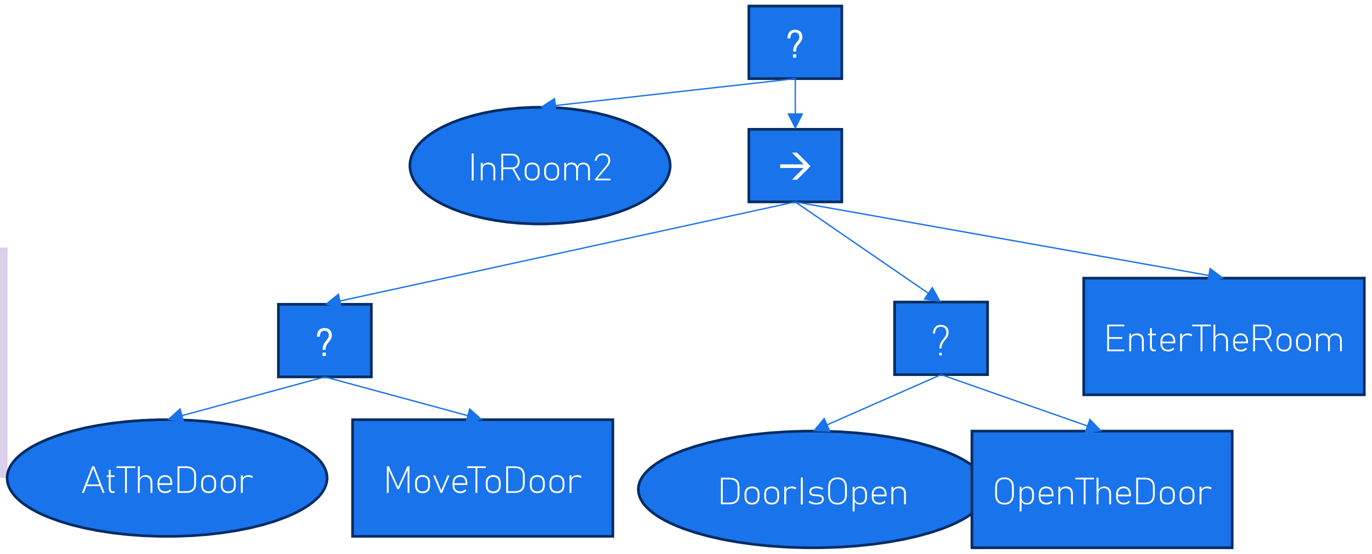
Behavior Trees

The execution of a BT starts from the root which sends ticks with a certain frequency to its child.
A **tick** is an **enabling signal** that allows the execution of a child.

When the execution of a node in the BT is allowed, it returns to the parent a status:
running if its execution has not finished yet,
success if it has achieved its goal, or
failure otherwise.



Behavior Tree



Nodes:

- Root node – Always a controller (general behavior). No parent, one child (ticks)
- Composite node ("Control flow ") –

  Selector: Executes children in order until one succeeds.

  Sequencer: Executes children in sequence until one fails.

  Parallel: Executes children simultaneously. One parent, and one or more children
- Leaf node ("Execution") – Actions or conditions. One parent, no child (Leaves)
- Decorator node ("Operator") – One parent, one child

Behavior Tree

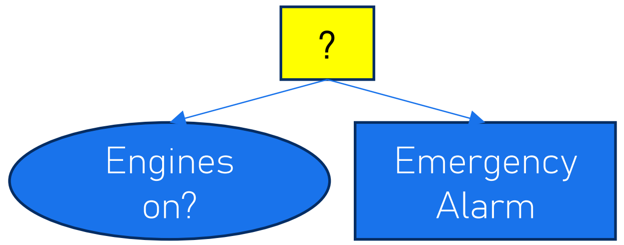
A **composite node** can have one or more children.

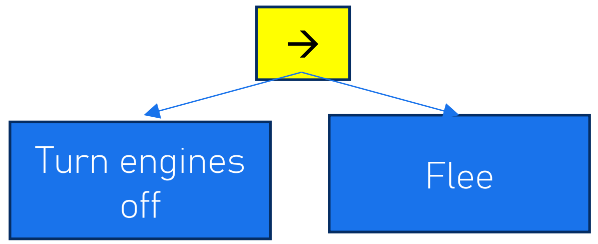
The node is responsible to propagate the tick signal to its children, respecting some order (flow control) ·

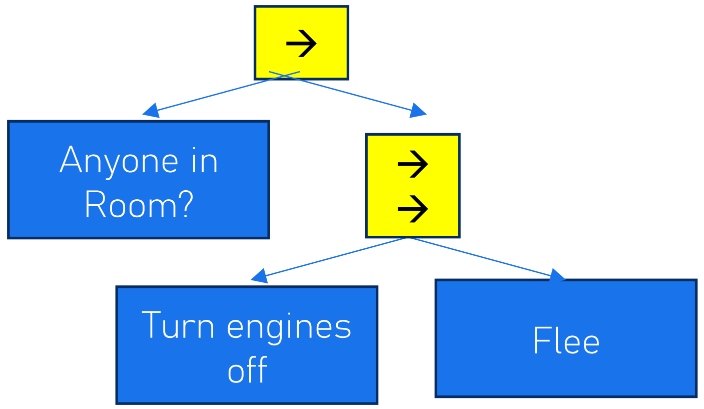
?
- **Selector** – Executes children in order <u>until one succeeds</u>. Returns SUCCESS when it finds the <u>first child</u> that returns SUCCESS. The **priority** node (sometimes called selector) ticks its children sequentially until <u>one of them</u> returns SUCCESS, RUNNING or ERROR. If all children return the failure state, the priority also returns FAILURE.

→
- **Sequence – A** node that returns SUCCESS **if** <u>all its children</u> return SUCCESS. Executes children in order until one fails.

→
→
- The **parallel** node ticks all children <u>at the same time</u>, allowing them to work in parallel. Returns SUCCESS if <u>all or some of its children</u> returns SUCCESS







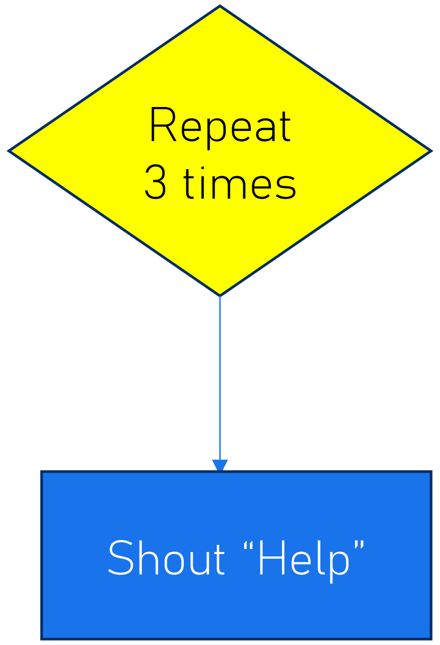
Behavior Tree

Decorators

are special nodes that can have only a single child. .

The goal of the decorator is to modify the behavior of the child by manipulating the returning value or changing its ticking frequency

- Repeater
- Inverter

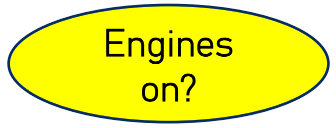


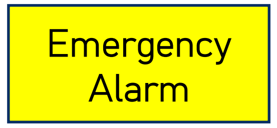
Behavior Tree

The **leaf nodes** **(aka)** Execution nodes are the primitive building blocks of the behavior tree.

They perform some computation and return a state value (functional)

- A **condition node** checks whether a certain <u>condition has been met</u> or not (e.g. “obstacle distance’”)
- An **action node** performs computations to <u>change the agent state</u> (e.g., the actions of a robot may involve sending motor signals)





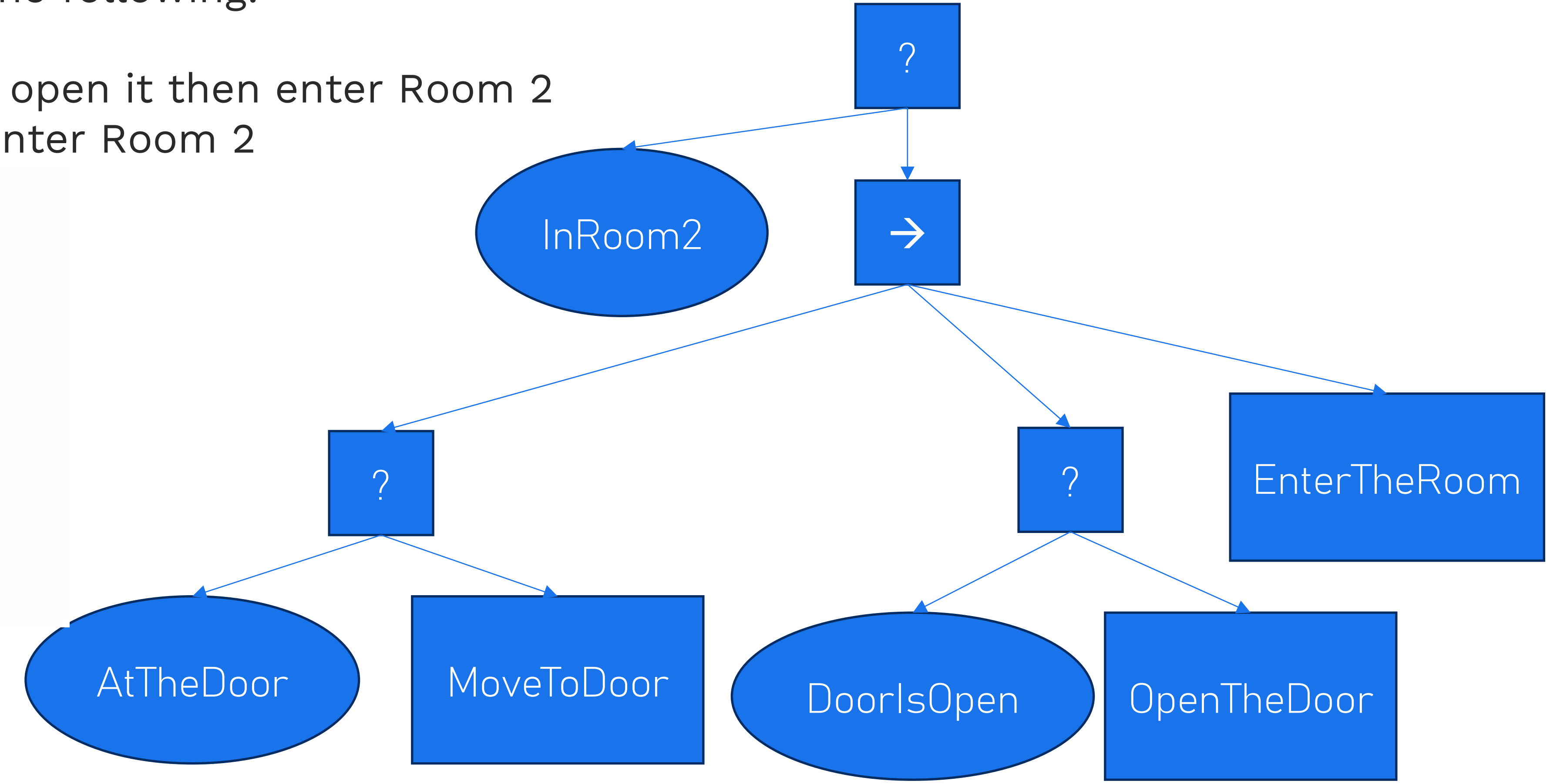
Behavior Trees

Node Status

- When ticked, a node executes its program or propagate the Tick signal if it is a Control Flow node. A node has to return status to its parent. The status is either one of these below:
- SUCCESS: when it has completed the job successfully: when a criterion has been met by a condition node or an action node has been completed successfully;
- FAILURE: when it has failed to execute the job: when a criterion has not been met by a condition node or an action node could not finish its execution for any reason;
- RUNNING: when the job is still running (or, being executed by Execution Platform): when an action node has been initialized but is still waiting the its resolution.
- ERROR: when some unexpected error happened in the tree, probably by a programming error (trying to verify an undefined variable). Its use depends on the final implementation of the leaf nodes.

Symbol	Node Type	Behavior
→	Sequence	Executes children in order until one fails
?	Selector	Executes children in order until one succeeds

The robot has to do the following:
- Move to the door
- If the door is closed, open it then enter Room 2
- If the door is open, enter Room 2



https://towardsdatascience.com/designing-ai-agents-behaviors-with-behavior-trees-b28aa1c3cf8a/

```
public class SimpleEnterRoom : MonoBehaviour
{

public bool inRoom2 = false;
public bool atTheDoor = false;
public bool doorIsOpen = false;

void Update()
{
    if (inRoom2)
    {
        Debug.Log("Already in the room!");
        return;
    }
    if (!atTheDoor){
        MoveToDoor();
        return;
    }
    if (!doorIsOpen)
    {
        OpenTheDoor();
        return;
    }
    EnterTheRoom();
}

void MoveToDoor()
{
    Debug.Log("Moving to the door...");
    atTheDoor = true;
}

void OpenTheDoor()
{
    Debug.Log("Opening the door...");
    doorIsOpen = true;
}

void EnterTheRoom()
{
    Debug.Log("Entering the room...");
    inRoom2 = true;
}
}
```

Behavior Tree

Exercise 4

Create Behavior Tree for NPC in Unity:

If you are threatened → run away

If low battery → recharge

Otherwise, → explore (walk + scan).

Benefits of BT

Maintainability: transitions in BT are defined by the structure, not by conditions inside the states. Nodes can be designed independent from each other, thus, when adding or removing new nodes (or even subtrees) in a small part of the tree, it is not necessary to change other parts of the model.

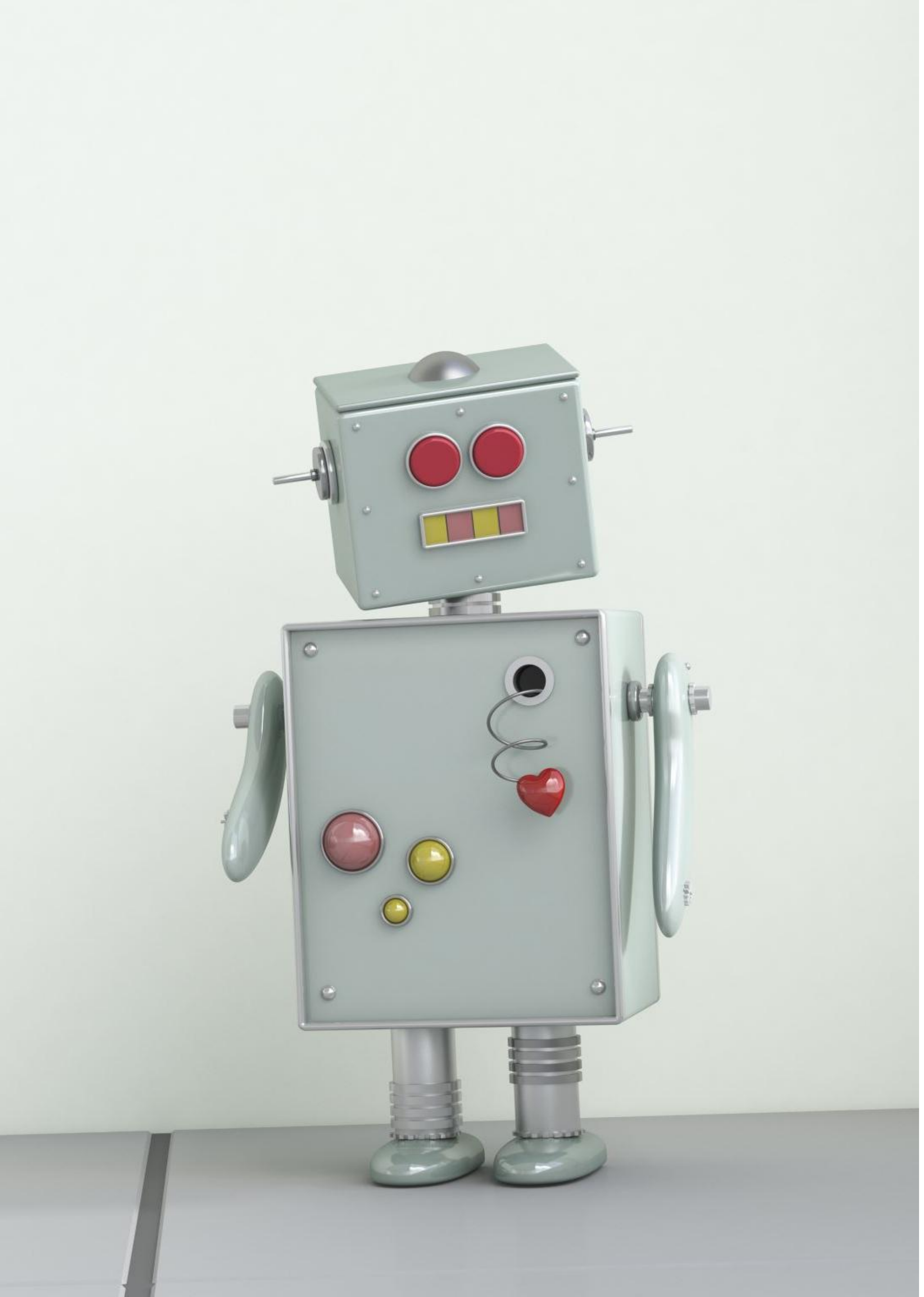
Scalability: when a BT have many nodes, it can be decomposed into small sub-trees saving the readability of the graphical model.

Reusability: due to the independence of nodes in BT, the subtrees are also independent. This allows the reuse of nodes or subtrees among other trees or projects.

Difficulties

- More complex to understand and design initially. It needs a well-thought-out hierarchical structure to avoid redundancy.
- It can have higher computational overhead in very large cases.

Technique	Advantages	Limitations
• Ad-hoc	• Simple, fast	• Not very scalable
• FSM	• Visual, intuitive	• State explosion
• Decision Tree	• Hierarchical structure	• Difficult to maintain large trees
• Behavior Tree	• Scalable, modular	• More complex implementation



Discussions

Imagine an **explorer robot** that needs to:

- Explore area (walk, scan)
- Avoiding obstacles
- Recharge battery when low level
- Flee if you detect a threat

FSM would be:

Many states: exploring, avoiding, recharging, running away, etc.

Complex transitions for different combinations.

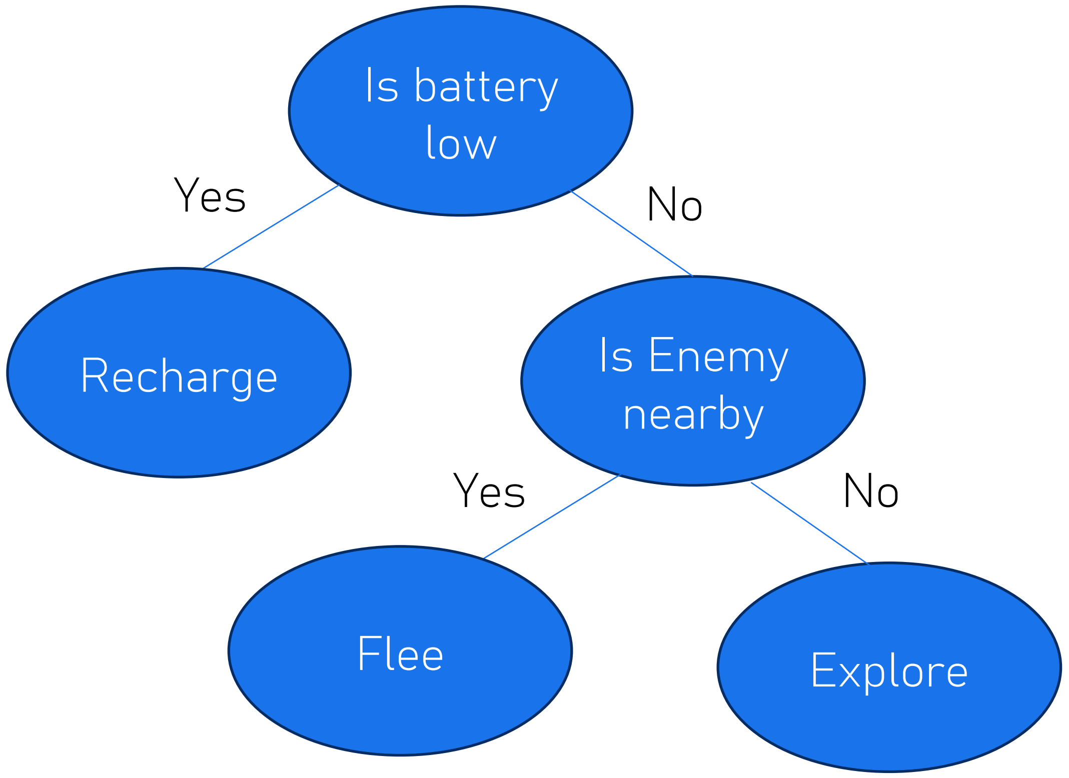
Decision Tree would be:

Check battery? If it goes down, it recharges

Checks for threat? If so, run away

If not, explore

But without a clear notion of sequencing.



Behavior Tree would be:

? Root (Selector)

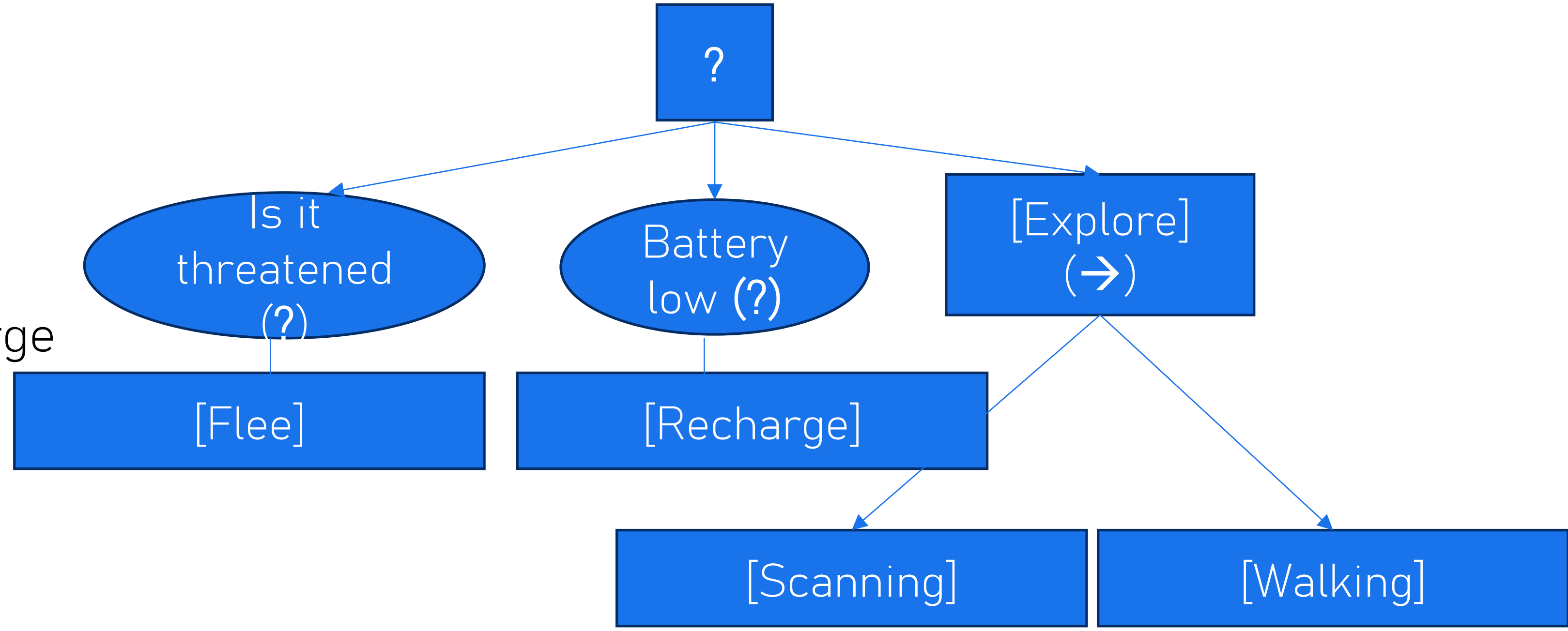
├── ? Is it threatened? → Flee

├── ? If battery is low? → Recharge

└── → Explore (Sequencer)

├── Scanning

└── Walking



Conclusion:

In practice, many autonomous agents use a hybrid combination of methods:

FSMs to control global agent modes.

Behavior trees for detailed local actions.

Decision trees for quick decision-making within a behavior.

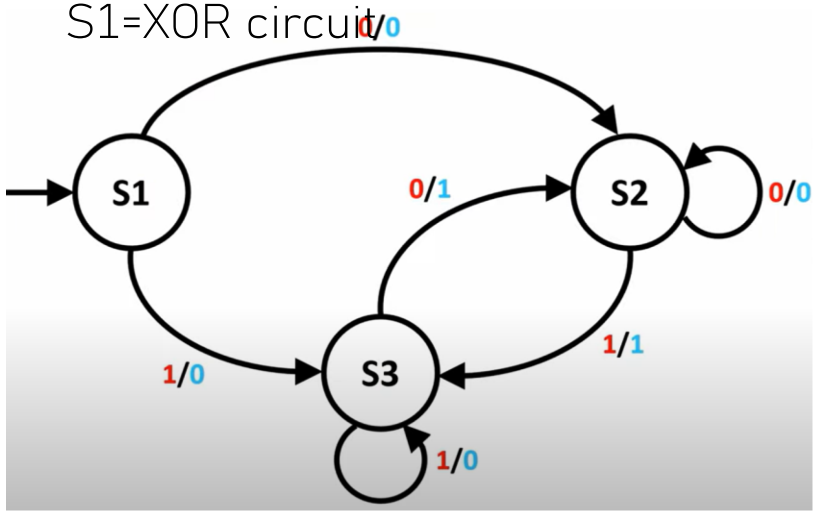
Machine learning to adjust parameters or choose actions.

Special cases:

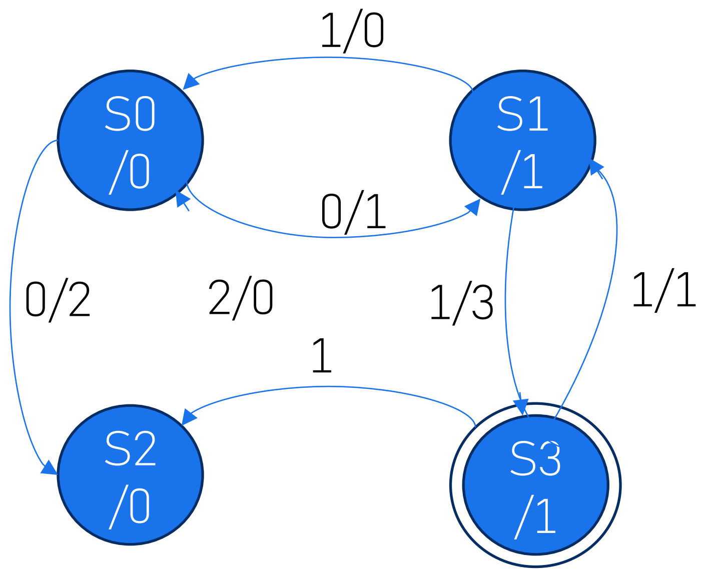
- **Mealy machine**
- **Moore machine**

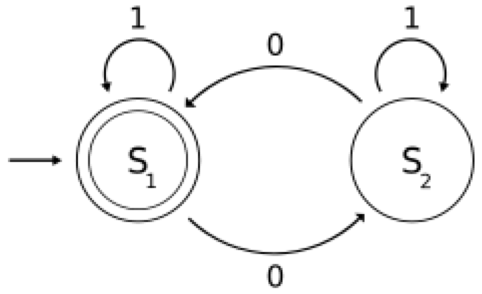
Feature	Mealy Machine	Moore Machine
Output depends on	Current **state + input**	**Only** current **state**
Output changes	Immediately when input changes (more responsive)	Only when entering a new state
Number of states	Often **fewer** (can reuse states for different inputs)	Often **more** (each output requires its own state)
Output function	Z=f(S,X)	Z=f(S)
Example	Vending machine that outputs based on inserted coin **and** button pressed	Elevator indicator showing floor number based only on **current state** (floor)

S1=XOR circuit

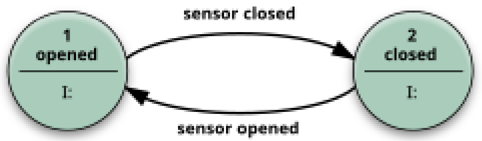


Mealy Machine (Output depends on state + input)
Output values (after the slash) are **on the arrows (transitions)**





Acceptor FSM; this example shows one that determines whether a binary number has an even number of 0s, where S_1 is an *accepting state* and S_2 is a *non accepting state*.



Transducer FSM: Mealy model example

Acceptors

(also called *detectors* or **recognizers**)

no output, make decision of final state, indicating whether or not the received input is accepted.

Each state of an acceptor is either accepting or non accepting.

Once all input has been received, if the current state is an accepting state, the input is accepted; otherwise it is rejected

Transducers

produce **output** based on a given input and/or a state using actions.