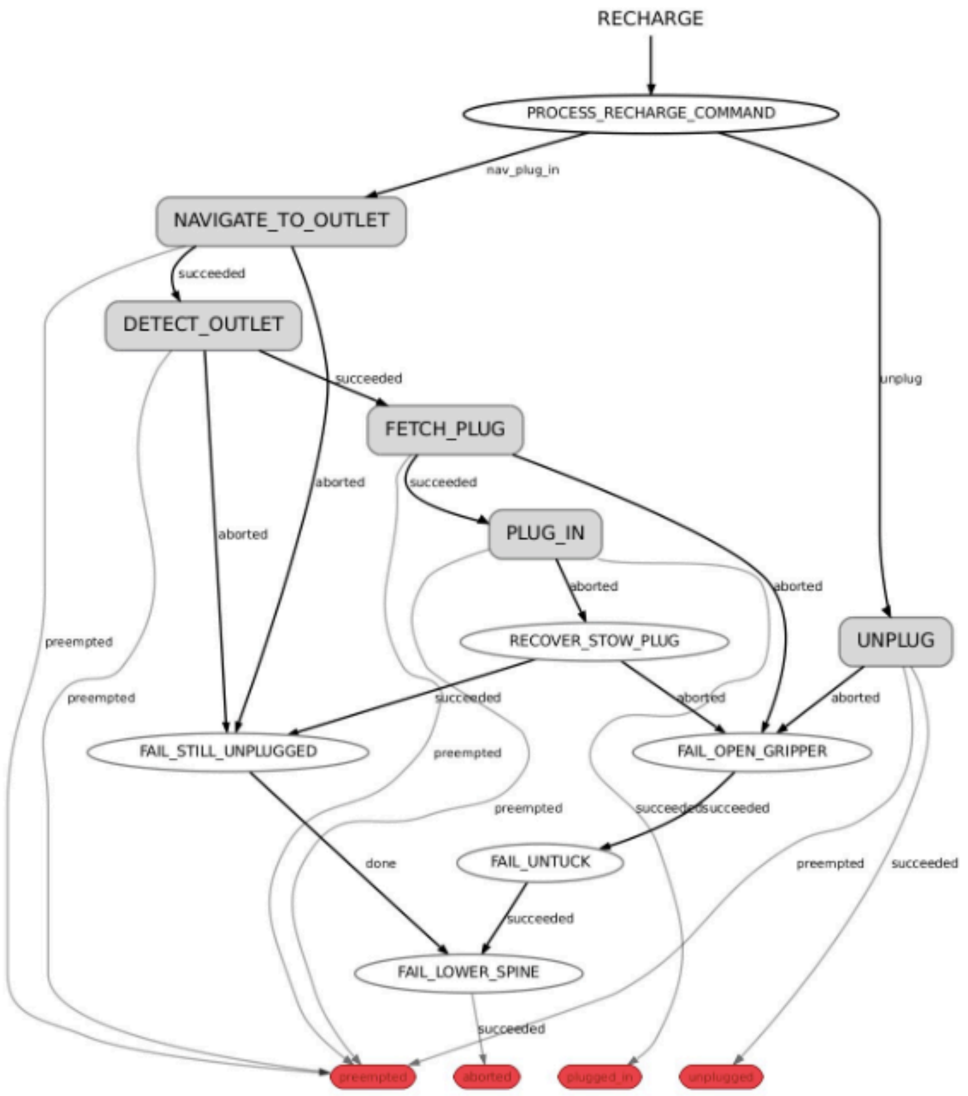
2.4 Hierarchical Finite State Machine (HFSM) & Behavior Tree (BT)

Problems of FSM

- N states -> N x N possible Transitions
- N can be very large
- And NxN is even larger

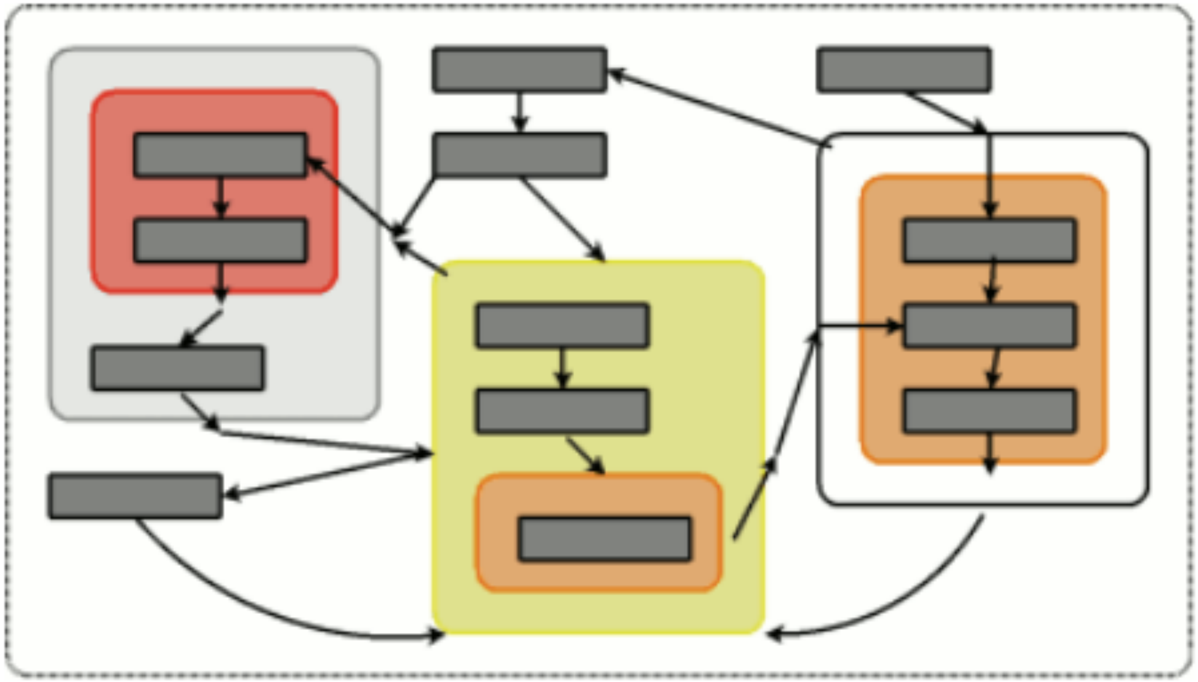


Other Problems with FSM

- **Maintainability**: when adding or removing a state, it is necessary to change the conditions of all other states that have transition to the new or old one. Big changes are more susceptible to errors that may pass unnoticed.
- **Scalability**: FSMs with many states lose the advantage of graphical readability, becoming a nightmare of boxes and arrows.
- **Reusability**: as the conditions are inside the states, the coupling between the states is strong, being practically impossible to use the same behavior in multiple projects.

Hierarchical Finite State Machine

- a.k.a StateCharts (first introduced by David Harel)



Harel’s StateCharts

- **Super-states** : groups of states.
  - These super-states too can have transitions, which allows you to prevent redundant transitions by applying them only once to super-states rather than each state individually.
- **Generalized transitions** : transitions between Super-states

Simplest Example

- Clustering / Super State

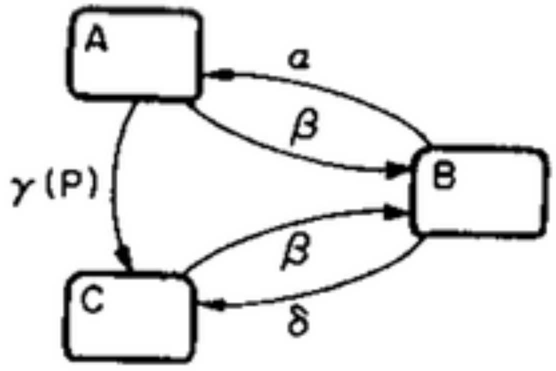


Fig. 1.

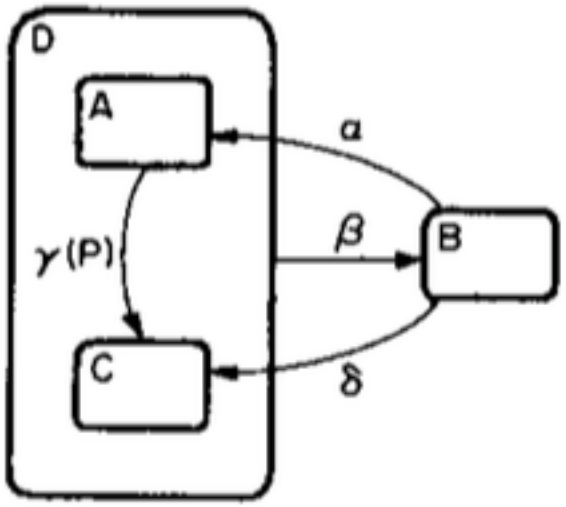
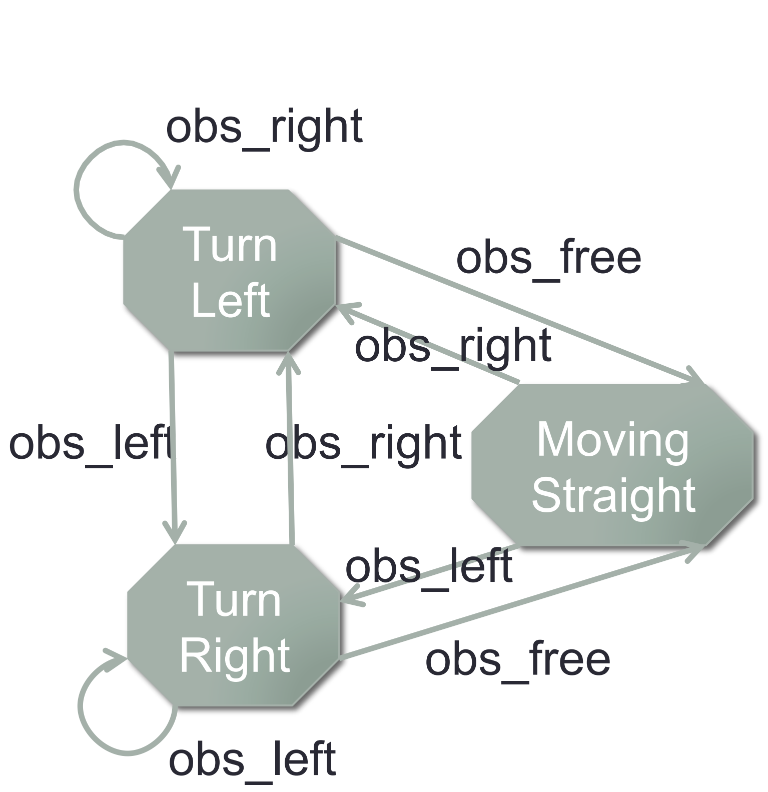
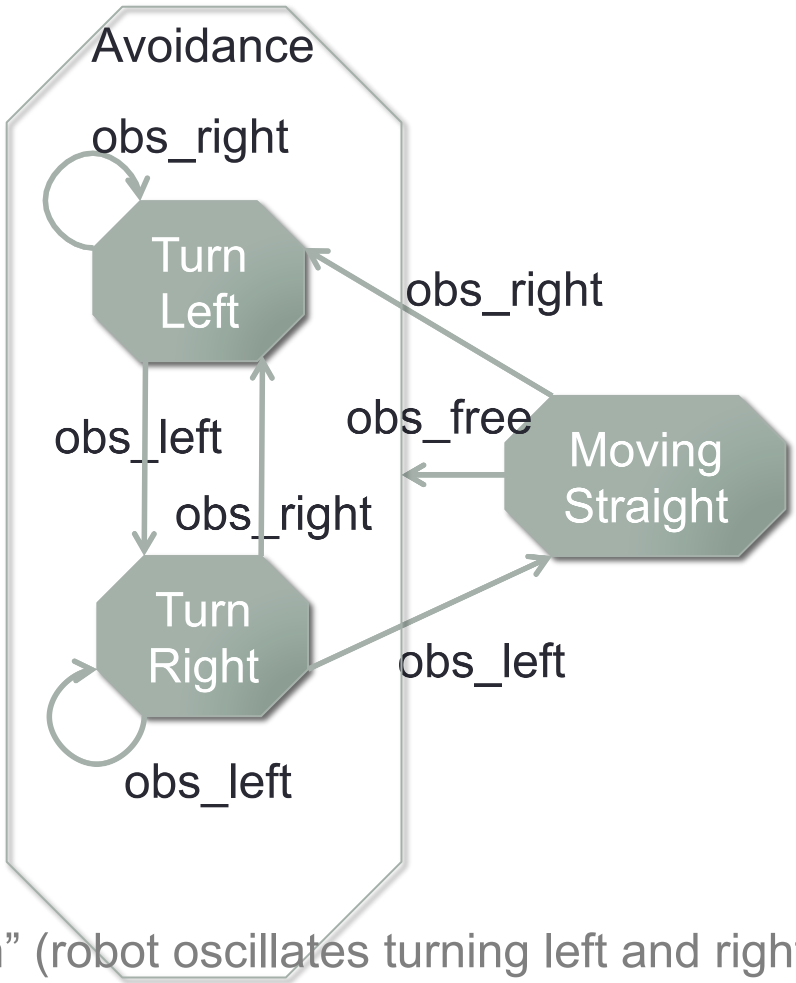


Fig. 2.

Obstacle Avoidance Example





Note: this algorithm can cause "oscillation" (robot oscillates turning left and right) in case of concave obstacle. But we discussed in class how to solve that

HFSM

- Refinement

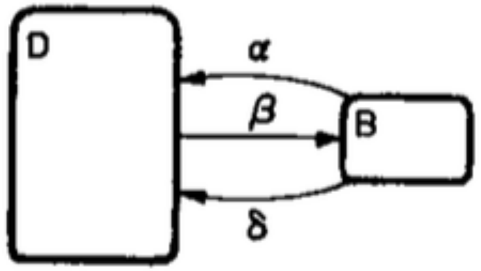


Fig. 3.

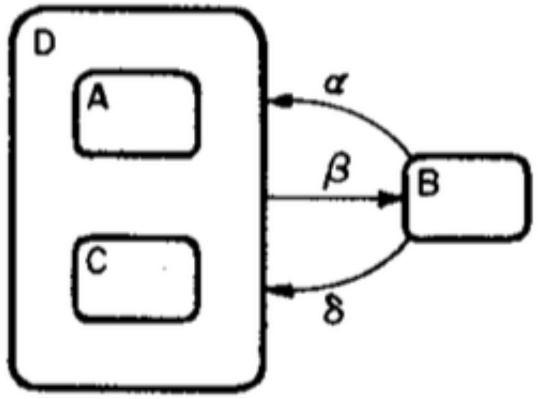


Fig. 4.

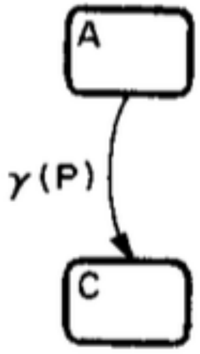


Fig. 5.

Behavior Inheritance

- HFSM combines hierarchy with programming-by-difference, which is otherwise known in software as *inheritance*.
- As class inheritance allows subclasses to *adapt* to new environments, behavioral inheritance allows substates to *mutate* by adding new behavior or by overriding existing behavior.
- State nesting introduces another fundamental type of inheritance, called *behavioral inheritance*

Behavior Inheritance

- Nested states can introduce new behavior by adding new state transitions or reactions (also known as internal transitions) for events that are not recognized by super-states. This corresponds to adding new methods to a subclass.

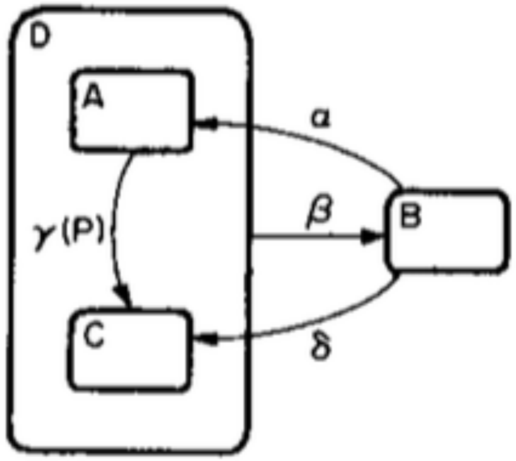


Fig. 2.

Good Enough?

HFSM certainly provide a way to reuse transitions, but it’s still not an ideal solution. The problem is that:

- Reusing transitions isn’t trivial to achieve, and requires a lot of thought when you have to create logic for many different contexts (e.g. dynamic goals, actor status).
- Editing transitions manually is rather tedious in the first place.
- Another solution is to focus on making individual states modular so they can be easily reused as-is different parts of the logic. Behavior trees take this approach_

Behavior Trees (BT)

- Mathematical Model of Plan Execution – describe switching between a finite set of tasks in a modular fashion
- Originated from Game Industry, as a powerful way to describe AI for “NPC”
  - Halo, Bioshock, Spore

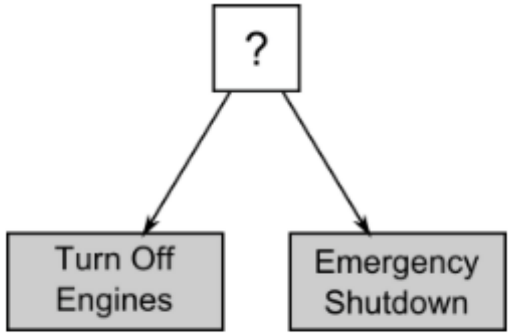
More Formally (Precisely)

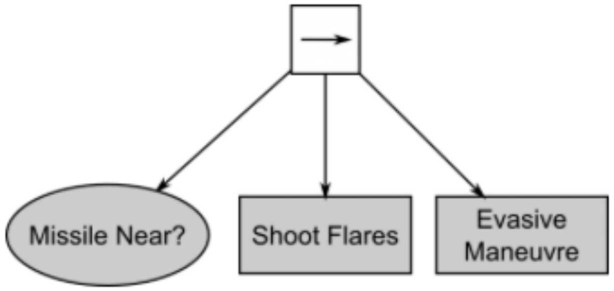
- Directed Acyclic Graph
- Four types of nodes:
  - **Root node** – no parent, one child (ticks)
  - **Composite node** ("Control flow ") – one parent, and one or more children
  - **Leaf node** ("Execution") – one parent, no child (Leaves)
  - **Decorator node** ("Operator") – one parent, one child

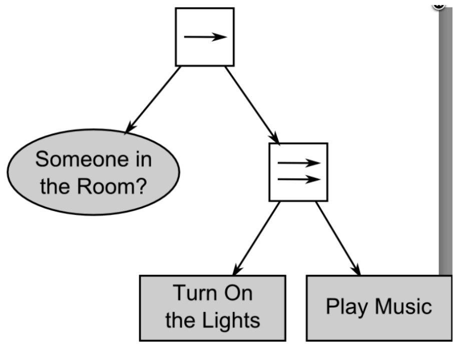
Composite Nodes

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)

- **The priority node** (sometimes called **selector**) ticks its children sequentially until one of them returns SUCCESS, RUNNING or ERROR. If all children return the failure state, the priority also returns FAILURE.
- **The sequence node** ticks its children sequentially until one of them returns FAILURE, RUNNING or ERROR. If all children return the success state, the sequence also returns SUCCESS.
- **The parallel node** ticks all children at the same time, allowing them to work in parallel.



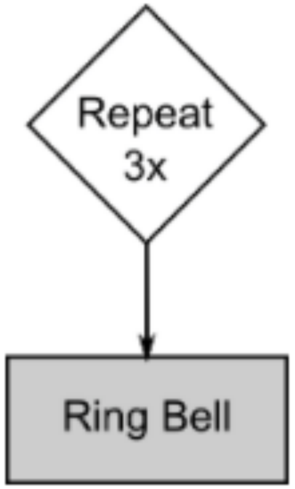




Decorator Nodes

Decorators are special nodes that can have only a single child. The goal of the decorator is to change the behavior of the child by manipulating the returning value or changing its ticking frequency

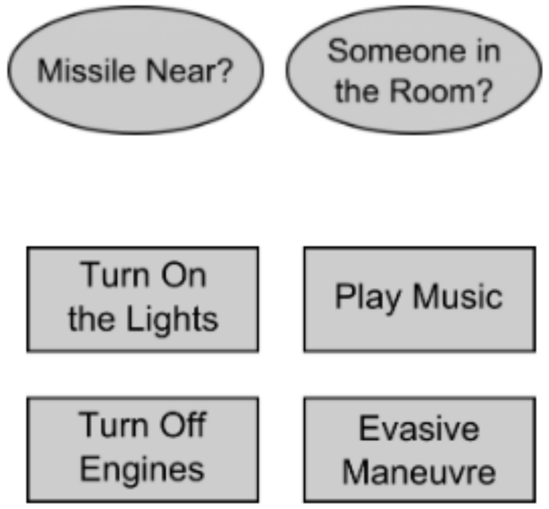
- Repeater
- Inverter



Leaf Nodes

The leaf 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 condition has been met or not (e.g. “obstacle distance’”)
- **An action node** performs computations to change the agent state (e.g., the actions of a robot may involve sending motor signals)



Node (State) Values

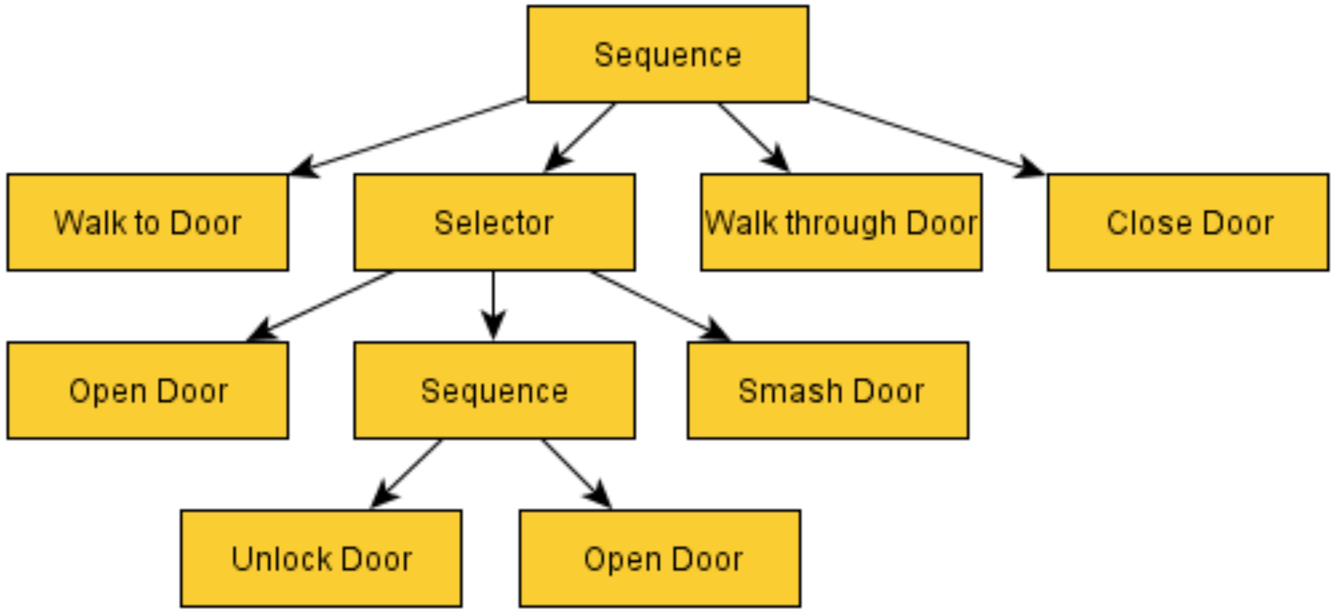
- SUCCESS: returned when a criterion has been met by a condition node or an action node has been completed successfully;
- FAILURE: returned when a criterion has not been met by a condition node or an action node could not finish its execution for any reason;
- RUNNING: returned when an action node has been initialized but is still waiting the its resolution.
- ERROR: returned 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.

Running of a BT

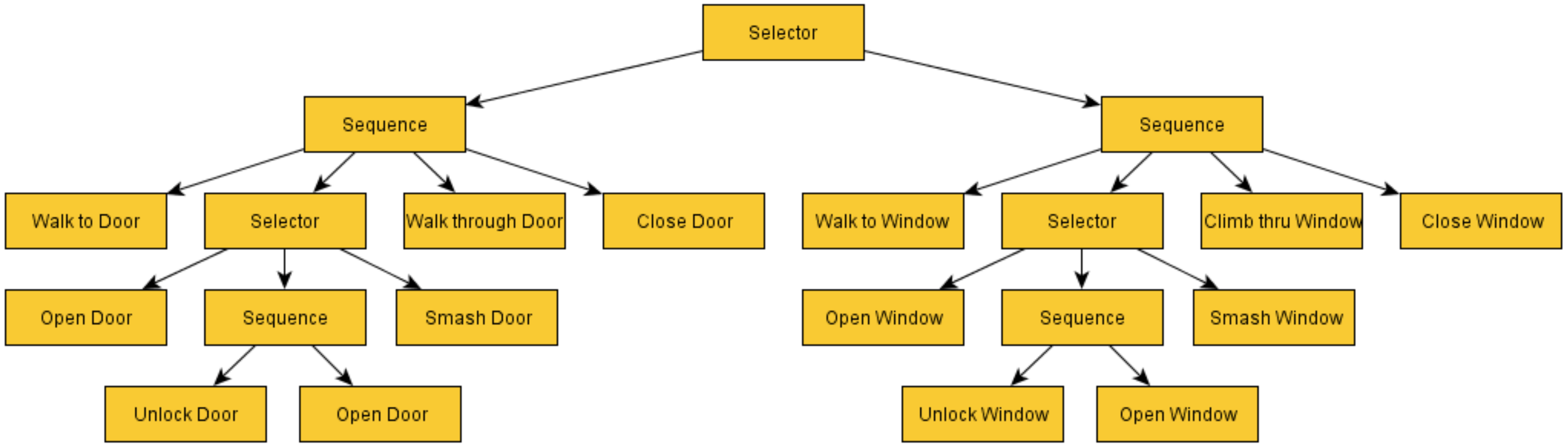
- The execution of a BT starts from the root which sends **ticks** with a certain frequency to its child. A tick is an enabling signal that allows the execution of a child. When the execution of a node in the BT is allowed, it returns to the parent a status **running** if its execution has not finished yet, **success** if it has achieved its goal, or **failure** otherwise.

BT Execution

- Depth-First Traversal



BT Execution



Tree Traversal Issue

- Always start from root node
- This isn't a very efficient way to do things, especially when the behavior tree gets deeper as its developed and expanded during development.
- Store any currently processing nodes so they can be ticked directly within the behavior tree engine rather than per tick traversal of the entire tree

Benefits of BT

- **Maintainability**: transitions in BT are defined by the structure, not by conditions inside the states. Because of this, 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.