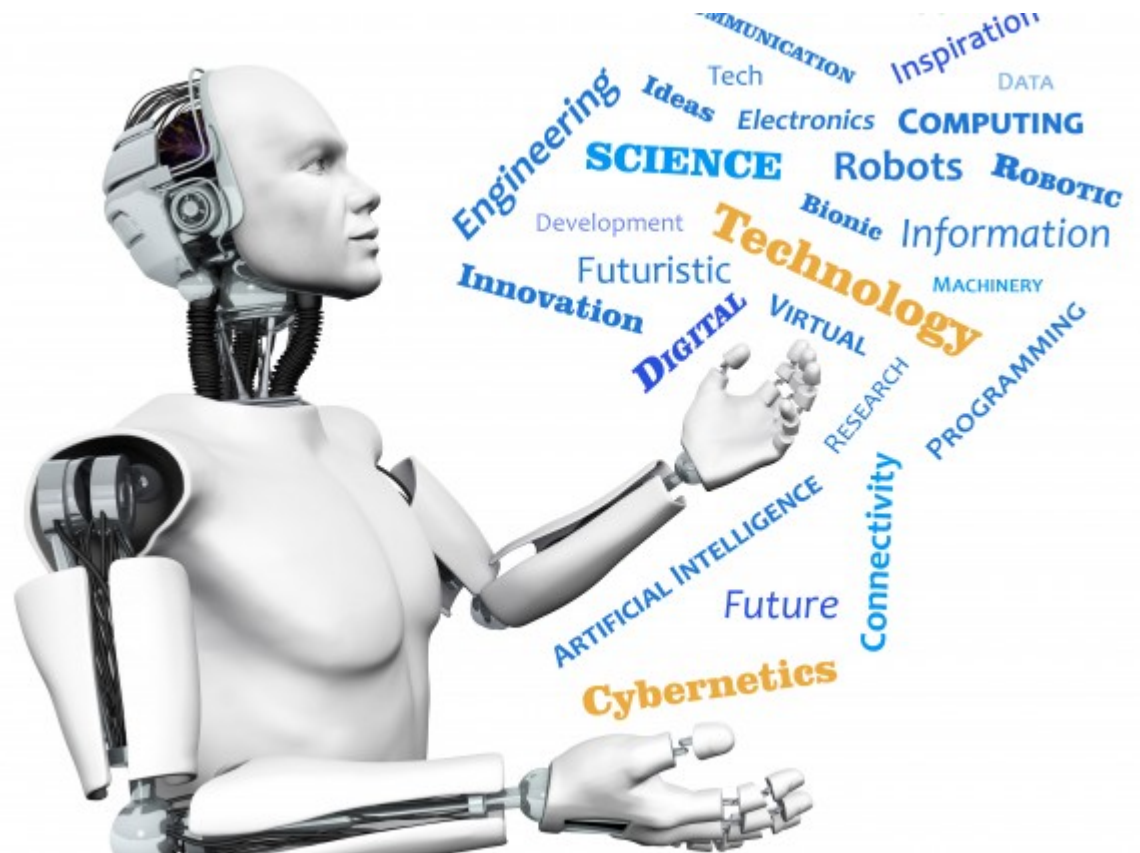


15-494/694: Cognitive Robotics

Dave Touretzky

Lecture 4:

Advanced State Machine
Concepts, and
Introduction to Particle
Filters



Differences From Classical FSMs

1. **Multi-State:**

- Multiple states can be active simultaneously (fork), and their completions can be synchronized (join).

2. **Hierarchical:**

- State machines can nest.

3. **Message Passing:**

- One state can send a message to another as part of a transition firing.

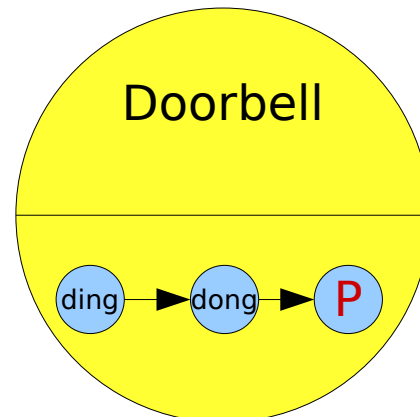
More On Hierarchy

- A nested state machine is started automatically when its parent node starts.
- The nested machine can cause its parent to signal *completion* by:
 - Transitioning to a `ParentCompletes` node
 - Calling `self.parent.post_completion()` from inside one of its nodes.
- Similarly for signaling parent *success* or *failure*: `ParentSucceeds` or `ParentFails`.

Nested State Machines

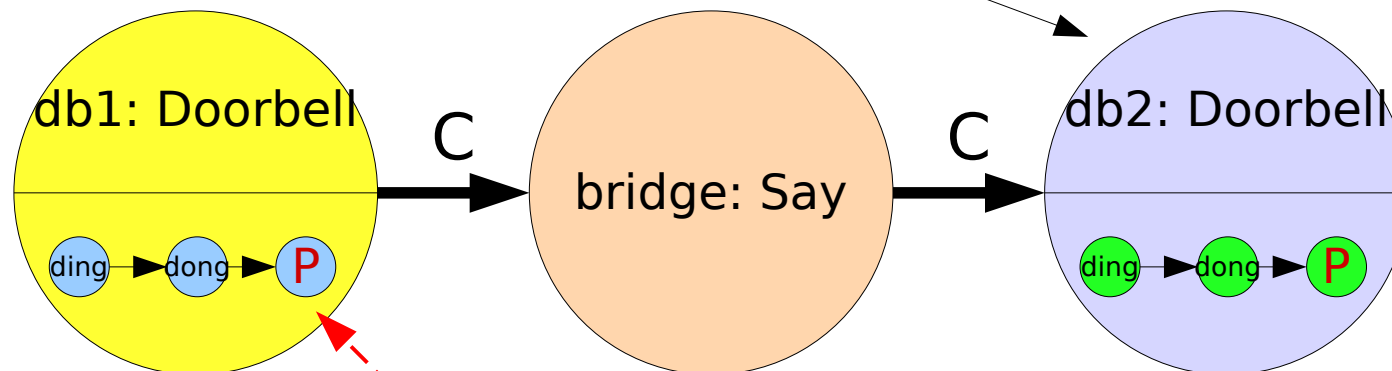
DingDong has an empty `start()` method, but it has a `setup()` method.

```
class Doorbell(StateNode):  
    $setup {  
        ding: Say('ding') =C=>  
        dong: Say('dong') =C=>  
        ParentCompletes()  
    }
```



Nested State Machines

```
class Nested(StateMachineProgram):  
    $setup {  
        db1: Doorbell() =C=>  
        bridge: Say('once again') =C=>  
        db2: Doorbell()  
    }
```



Will be triggered by db1's nested ParentCompletes node

Message Passing

- Nodes can signal “data events” that data transitions look for:

```
self.post_data(5)
```

- Transitions can match the data item:

```
foo =D(5)=> draw_pentagram
```

```
foo =D(6)=> draw_hexagram
```

- Transitions can also do wildcard match:

```
foo =D=> draw_stuff
```

Message Passing (cont.)

- When a transition activates a node, the node's start method is passed the event that triggered the transition.
- If this was a DataEvent, the start method can extract the data item and process it.

Sending Data

```
class Sender(StateNode):  
  
    def start(self, event=None):  
        super().start(event)  
        value = random.random()  
        self.post_data(value)
```


Receiving Data

```
class Receiver(StateNode):  
  
    def start(self, event=None):  
        super().start(event)  
        if isinstance(event, DataEvent):  
            value = event.data  
            print('Value received:', value)
```

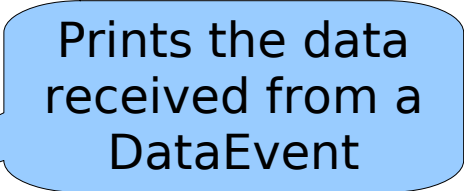
Sending and Receiving

```
class SendRecv(StateMachineProgram):  
    $setup{  
        Sender() =D=> Receiver()  
    }
```

```
C> runfsm('SendRecv')  
Value received: 0.380313711
```

Iteration

```
class IterDemo(StateMachineProgram) :  
    $setup{  
        loop: Iterate(4)  
        loop =D=> Print() =Next=> loop  
        loop =C=> Print('Done!')  
    }
```



Use =CNext=> instead of =Next=> to wait for completion.

Default Transitions

For data events and text message events, value matches take priority over defaults.

```
foo =TM( ' cat ' )=> Say( ' meow ' )
```

```
foo =TM( ' dog ' )=> Say( ' woof ' )
```

```
foo =TM=> Say( ' wacka-wacka ' )
```

How does this work? Default (wildcard) transitions have a slight time delay to allow any matching value transition to fire first.

Tap Events

- The SDK generates tap events when someone taps on a cube.
- We turn these into `cozmo_fsm TapEvents` that can be matched by a `=Tap=>` transition:
 - `=Tap(cube2)=>`
 - `=Tap=>`
- We need to check the tap intensity to reject false positives.

Face Events

- The SDK generates face events whenever a face is detected in the camera image.
- We turn these into `cozmo_fsm` `FaceEvents` that can be matched by a `=Face=>` transition:

`=Face('Dave')=>`

`=Face=>`

- Should probably provide separate cases for `FaceAppeared` and `FacePresent`.

The Event Loop

- While the SDK is connected to the robot and `simple_cli` is running, the value of `asyncio.get_current_event_loop()` is available in `robot.loop`.
- From `simple_cli`, in order to run a node we have to schedule it via this event loop.
- This is what the `now()` method does:
`Forward(50).now()`

Do It “Now”

```
class StateNode(EventListener):  
    ...  
    def now(self):  
        self.robot.loop.  
            call_soon(self.start)
```


EventListener

- Parent class of both StateNode and Transition.
- Includes a polling feature: an instance can request that its poll() method be called every t seconds.
- Polling begins when the instance's start() method is called and ends when stop() is called.

Uses of Polling

- DriveForward and DriveTurn use polling to check the robot's progress and decide when to stop.
- TimerTrans uses the polling interval to know when to fire.
- ArucoTrans uses polling to check if a marker has appeared in the camera image.

Animation and Trigger Nodes

- Animation nodes take an animation name as a string argument. There are over 900 to choose from.

```
AnimationNode('anim_bored_01')
```

- AnimationTriggerNodes take an `_AnimTrigger` object as an argument.

```
AnimationTriggerNode(  
    cozmo.anim.Triggers.  
    CubePouncePounceNormal  
)
```

Named Transitions

- A complex state machine may have a lot of CompletionTrans, SuccessTrans, and TimerTrans transitions.
- This makes the trace confusing: what is completiontrans5 doing?
- Solution: assign meaningful names to your transitions.

```
try_grab =grabbed:C=> open_it  
try_grab =fumbled:F=> reposition
```

Writing Your Own Transitions

- Rarely necessary, unless you're developing new robot functionality.
- How to do it:
 - `__init__()` to store constructor parameters.
 - `start()` to subscribe to events if needed.
 - `handle_event()` to examine the events and call `self.fire(event)` if needed.
 - `poll()` if polling is needed.

SeeBoth Transition

```
class SeeBoth(Transition):
    def __init__(self, thing1, thing2):
        super().__init__()
        self.thing1 = thing1
        self.thing2 = thing2
        self.set_polling_interval(0.1)

    def poll(self):
        if self.thing1.is_visible and
           self.thing2.is_visible:
            self.fire()
```

See12.fsm

```
class See12(StateMachineProgram) :
  $setup {
    StateNode()
      =SeeBoth(cube1, cube2) =>
        Say('I saw both')
  }
```

simple_cli 'show' commands

- show active
 - Shows the currently active nodes and transitions.
- show viewer
 - Shows the camera viewer
- show worldmap_viewer
 - Shows the worldmap viewer

Particle Filters

Intro to Particle Filters

- Odometry is unreliable.
 - Still useful for short trajectories.
 - But error accumulates quickly.
- Solution: use visual landmarks to correct for odometry error.
- But vision is unreliable too!
 - Landmark pose estimation is noisy.
 - Landmarks aren't always available.

Probabilistic Robotics

- Probabilistic robotics is based on the idea that we should embrace the noisiness.
- Instead of discrete values, think in terms of *probability distributions*.
- Robot's location is not (x,y) , but a *distribution* of possible locations, some more likely than others.

Modeling Location Distributions

- Particle filters are a way to model distributions.
- Think of each particle as a “guess” (hypothesis) about the robot's location.
- Assume we have a map with landmarks.
- Each guess predicts how the landmarks should look from that location.

Modeling Location Distributions

- Particles representing good guesses will accurately predict the landmark locations.
 - Good predictions earn a high weight.
- Bad guesses lead to poor predictions.
 - Poor predictions result in a low weight.
- As we accumulate sensor data, we can figure out which particles are the good guesses.

Particle Filter Demos

- [particle_filter_demo](#) is linked from the class schedule and can be found in the "Class/demos" directory.
- It shows the robot wandering in a maze; walls are landmarks.
- The robot's estimated location (white circle) is the average of all the particles.
- `pfdemo.py` is another demo you can try.

Resampling

- Bad guesses are a waste of resources.
- When we've accumulated enough data, we can generate a new set of particles to try to concentrate resources in the region of good guesses.
- Particles with high scores are chosen to spawn new particles.
- Low-scoring particles are unlikely to spawn.

Motion Model

- So far we have a robot that is standing still, receiving sensor data, and trying to figure out its location on the map.
- But the robot needs to move.
 - Stationary robots aren't useful.
 - Motion allows the robot to see more landmarks.

Motion Model (cont.)

- How can we accommodate motion?
- Solution:
 - As the robot moves, drag the particles along with it.
- But odometry is noisy!
 - Add noise (via a motion model) to the particle locations because we know that motion is unreliable.

SLAM

- What if we don't have a world map?
- SLAM: Simultaneous Localization And Mapping.
- Now each particle represents a slightly different map of the world, plus the robot's estimated location on that map.
- We will look at this in the next lecture.