15-494/694: Cognitive Robotics

Dave Touretzky

Lecture 3:

Finite State Machines and the aim_fsm Module

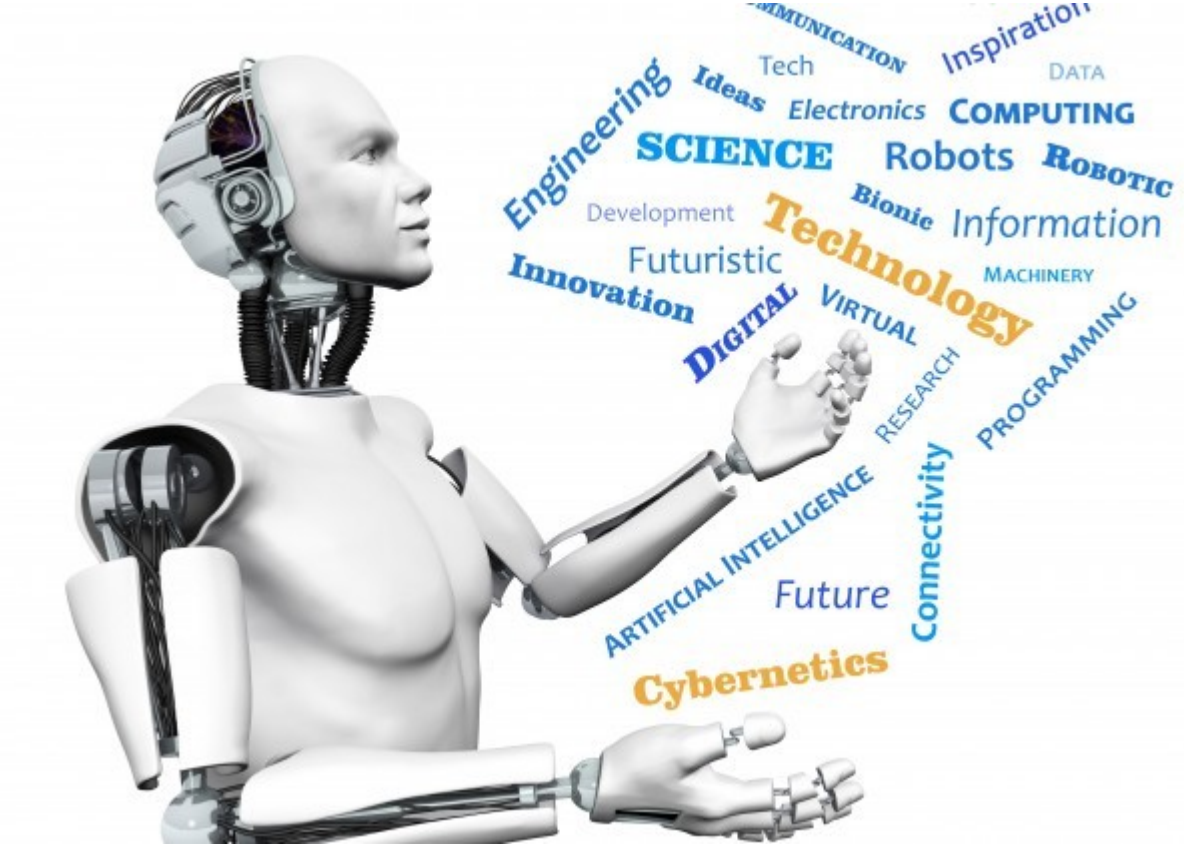


Image from http://www.futuristgerd.com/2015/09/10

Event Loop Recap

- Python's asyncio module provides an event loop (scheduler) but no events.
- Vex-aim-tools provides events and an event dispatcher.
- But programming is still cumbersome: sequencing must be hand-coded, since robot actions are asynchronous.
- Solution: state machines.

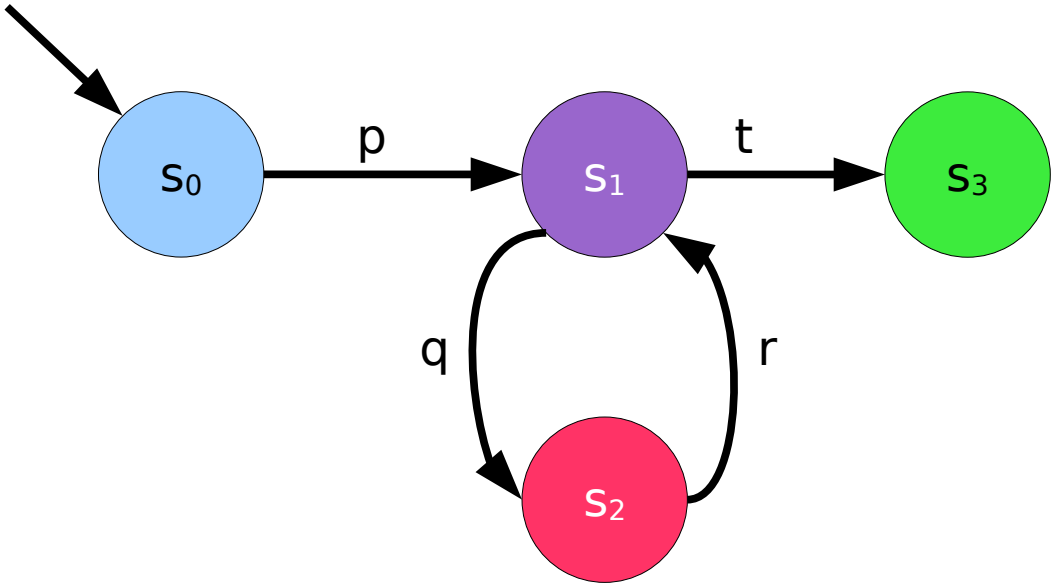
What Is A State Machine?

A classic finite state machine consists of:

- A finite set of discrete states $\{s_i\}$.
- A distinguished start state s_0.
- A set of transitions $\{ s_i \xrightarrow{c} s_j \}$.
- Each transition has a condition c that determines when the transition can apply.

State Machines Are Graphs

- The states are nodes.
- The transitions are labeled links.



FSMs in Robot Programming

- State machines are widely used in robot programming, from LEGO Mindstorms (NXT-G) to ROS (Smach).
- In robotics:
  - Nodes specify *actions*.
  - Transitions specify *reactions* (to events).
  - Events may be associated with an action, e.g., completion or failure.
  - Events can also be external, e.g., the user spoke, or tapped on the touch screen.

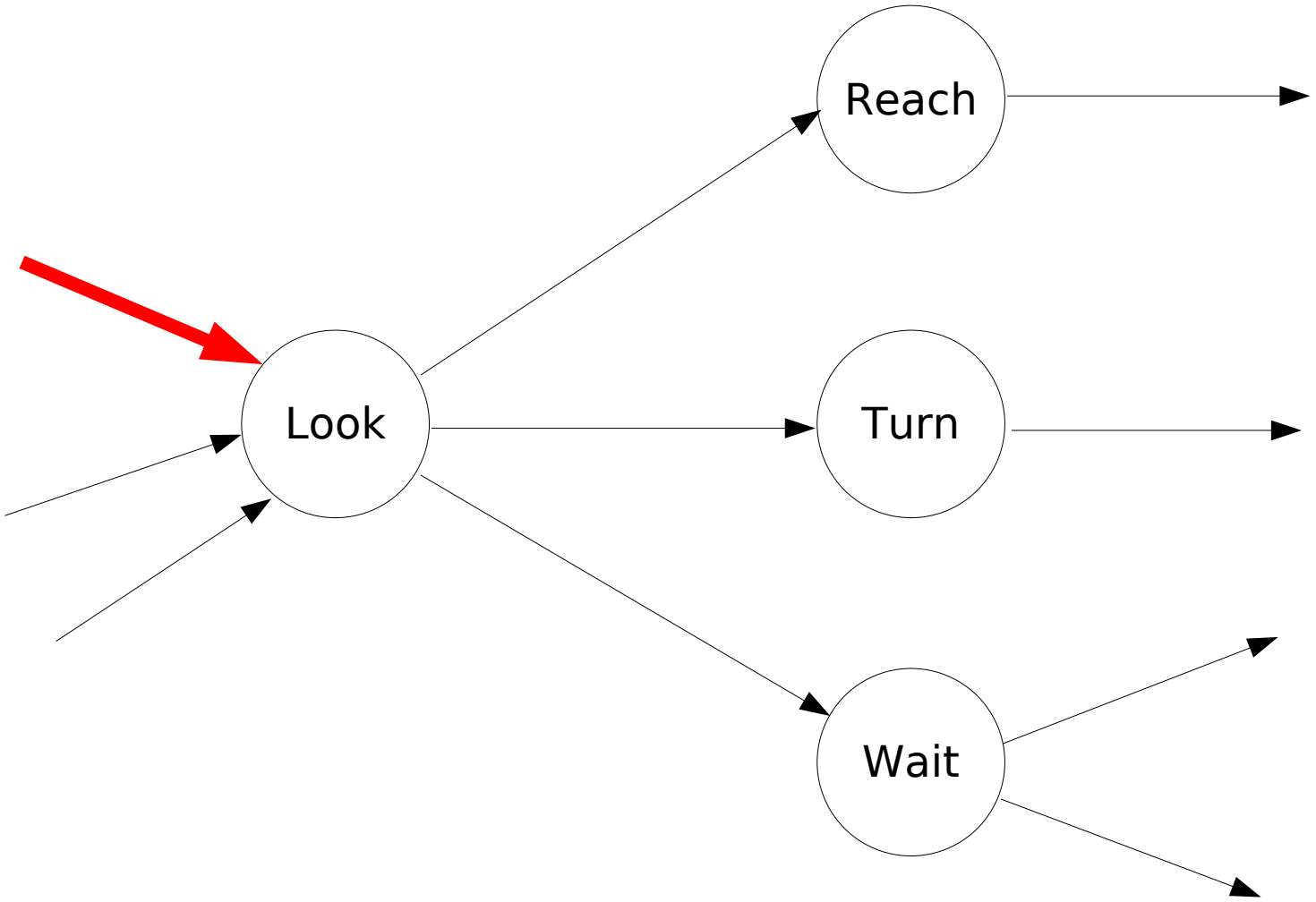
Advantages of FSMs

- Separates the control logic (links) from the functionality (nodes).
- The control logic can be expressed concisely as a graph.
- Provides an easy way to handle control problems such as:
  - fork/join
  - timeouts
- Easy way to trace execution.

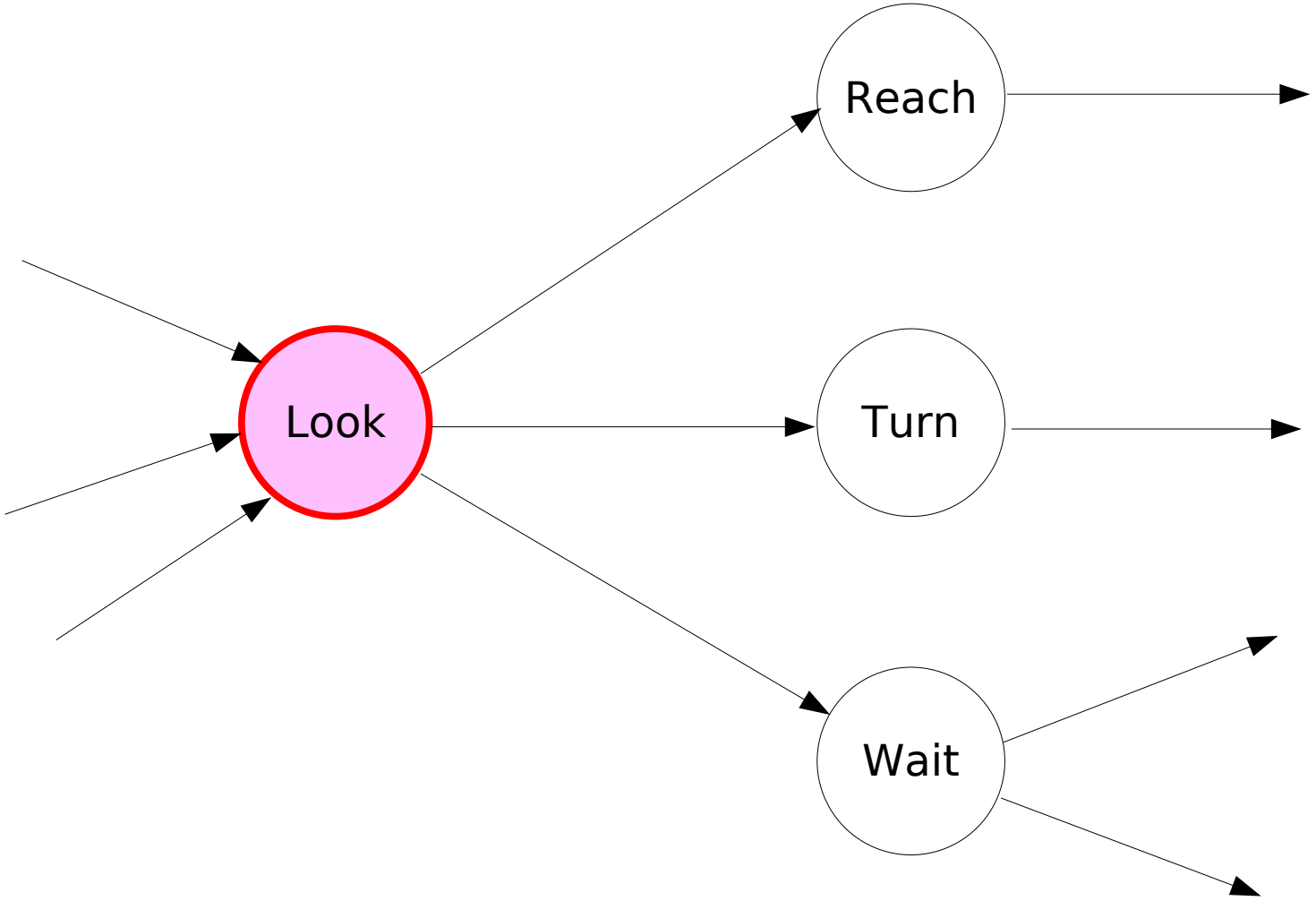
Event-Driven Architecture

- Robots typically use an event-driven architecture with many types of events.
  - Nodes can generate events by calling post_event().
  - Completing an action generates an event.
  - Sensors can also generate events.
- Transitions *listen for events* to determine when they should fire. (Nodes can also listen for events if they want to.)
- In aim_fsm, both StateNode and Transition are subclasses of EventListener.

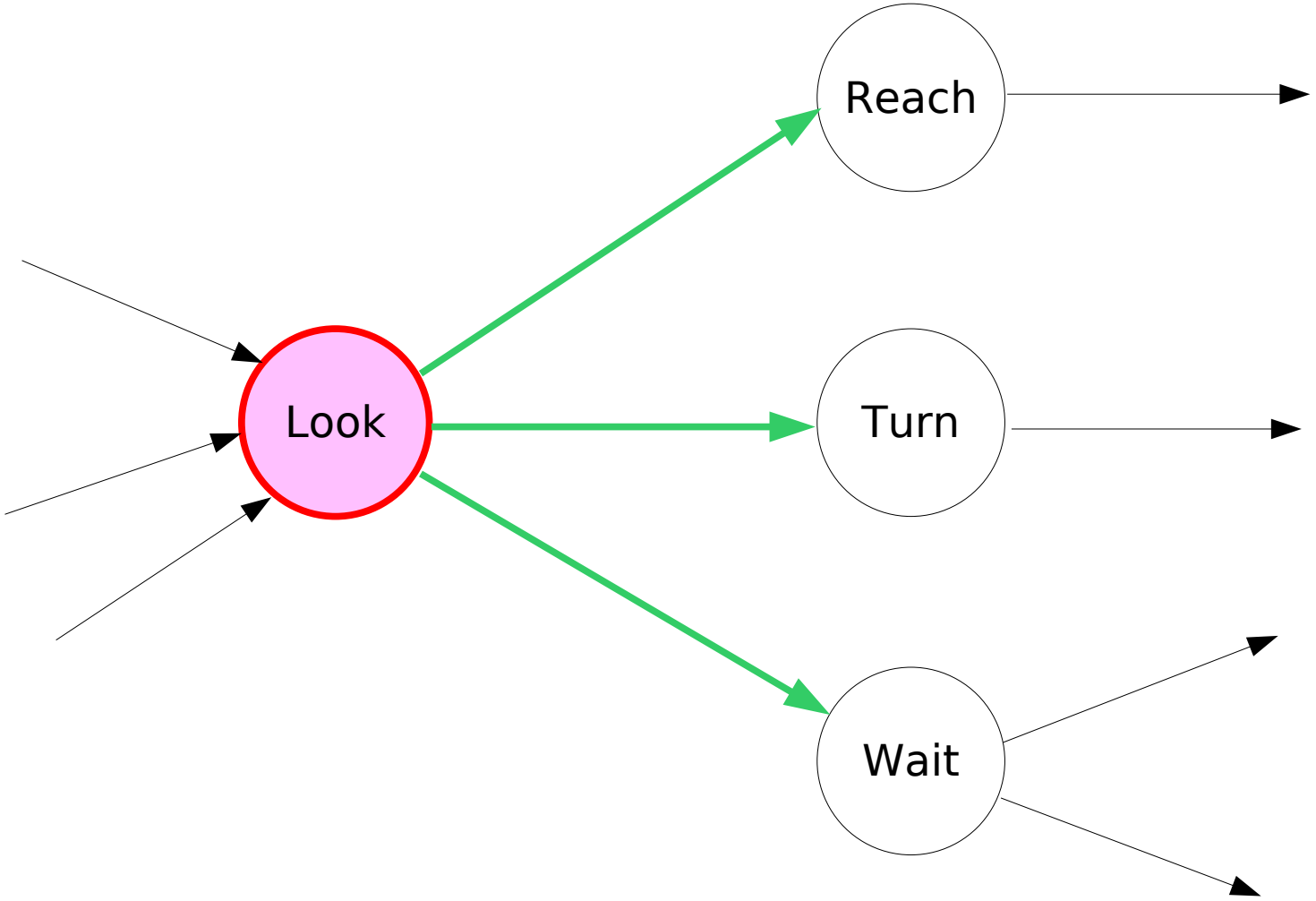
Transition firing activates state node Look.



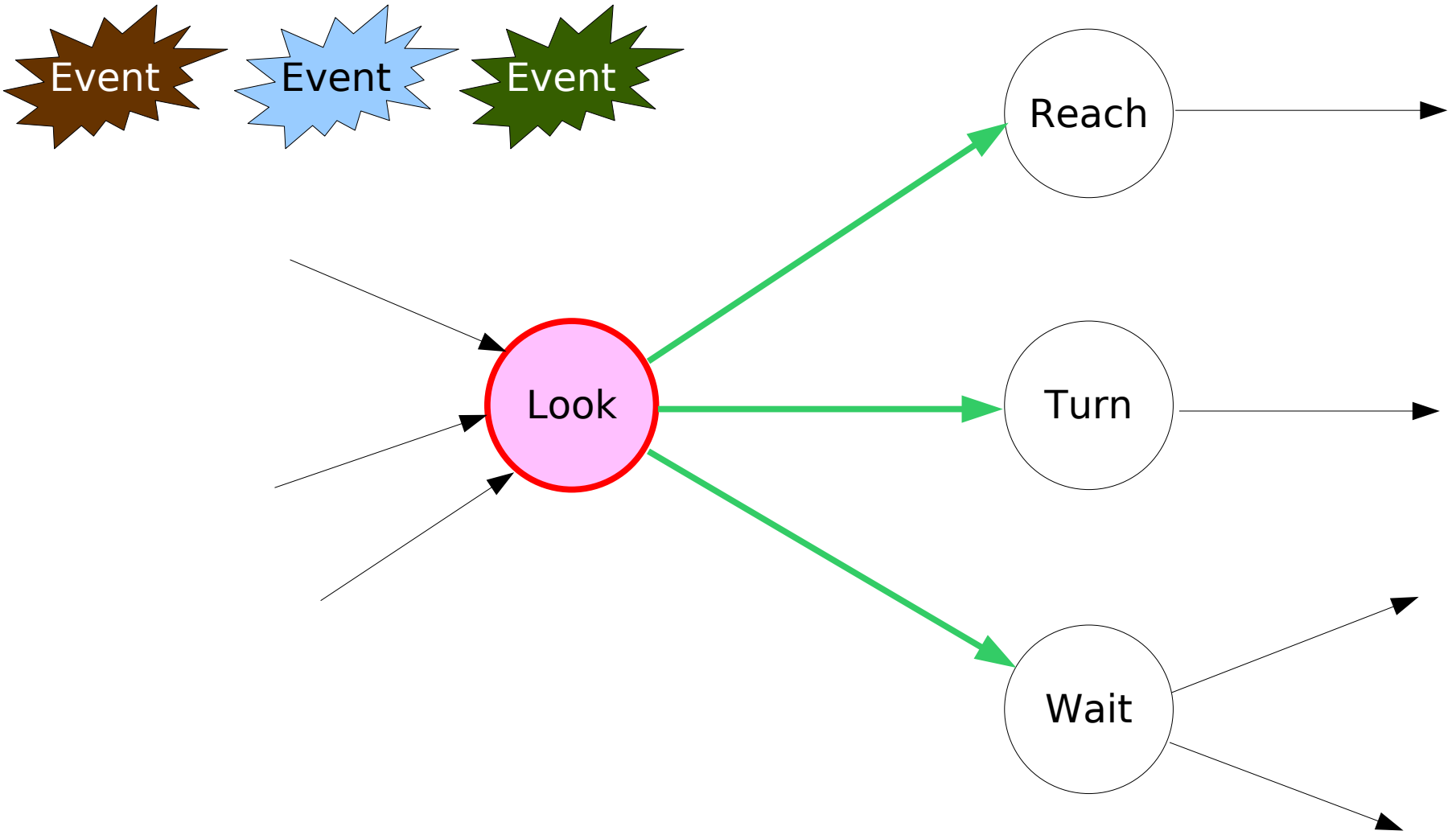
Look's start() method calls parent class StateNode's start() method.



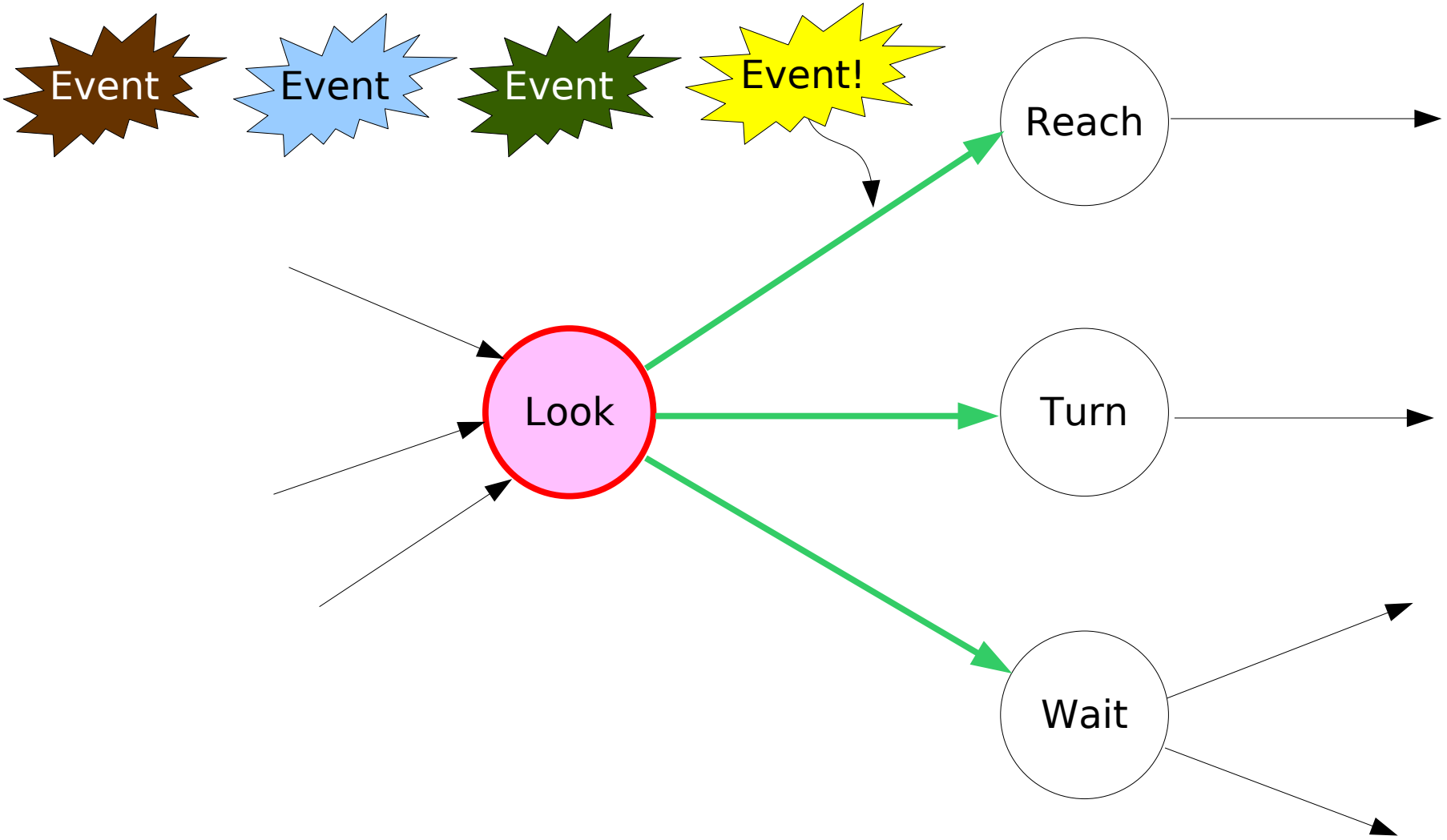
Look's outgoing transitions become active and begin listening for events.



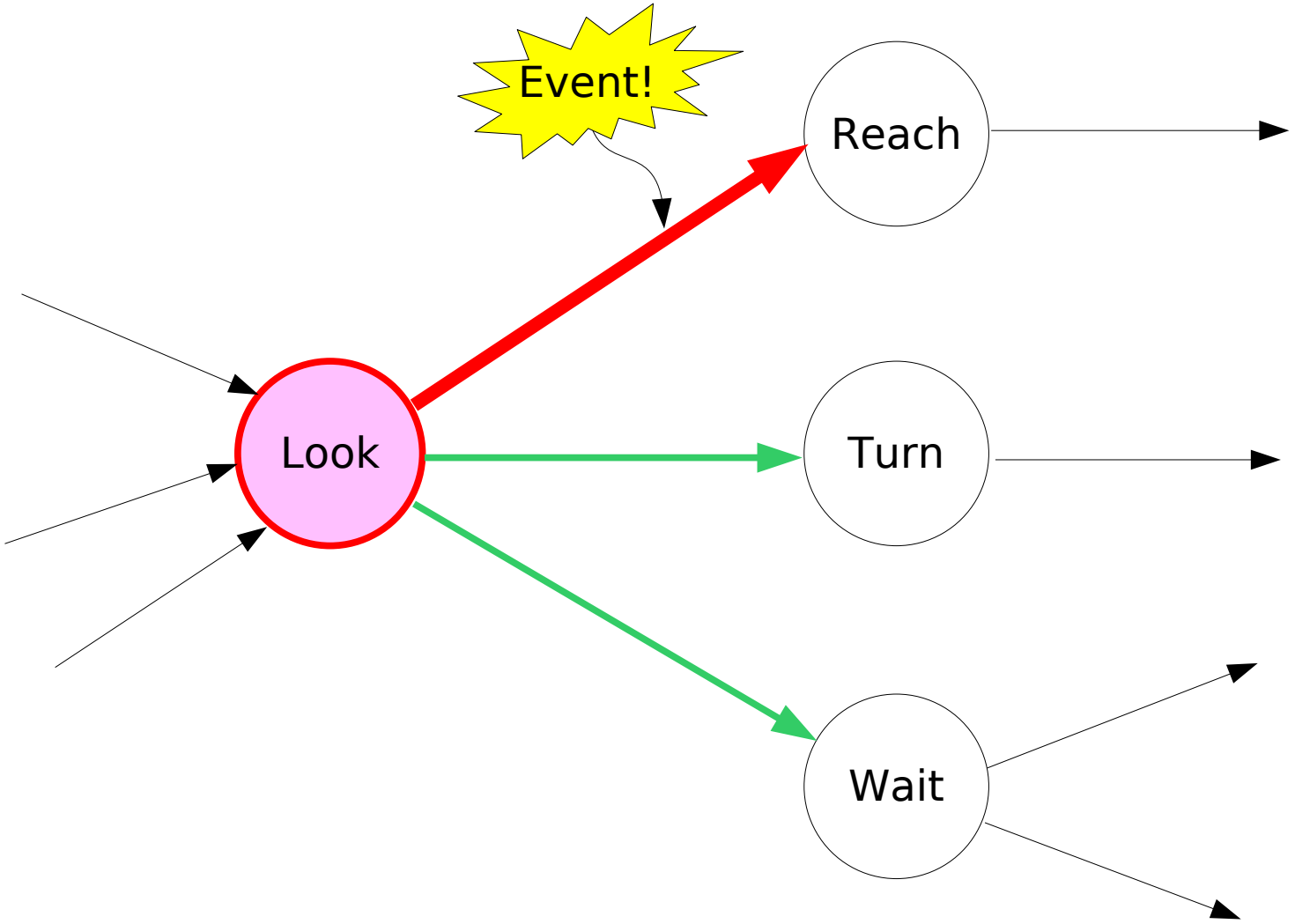
Random things happen....



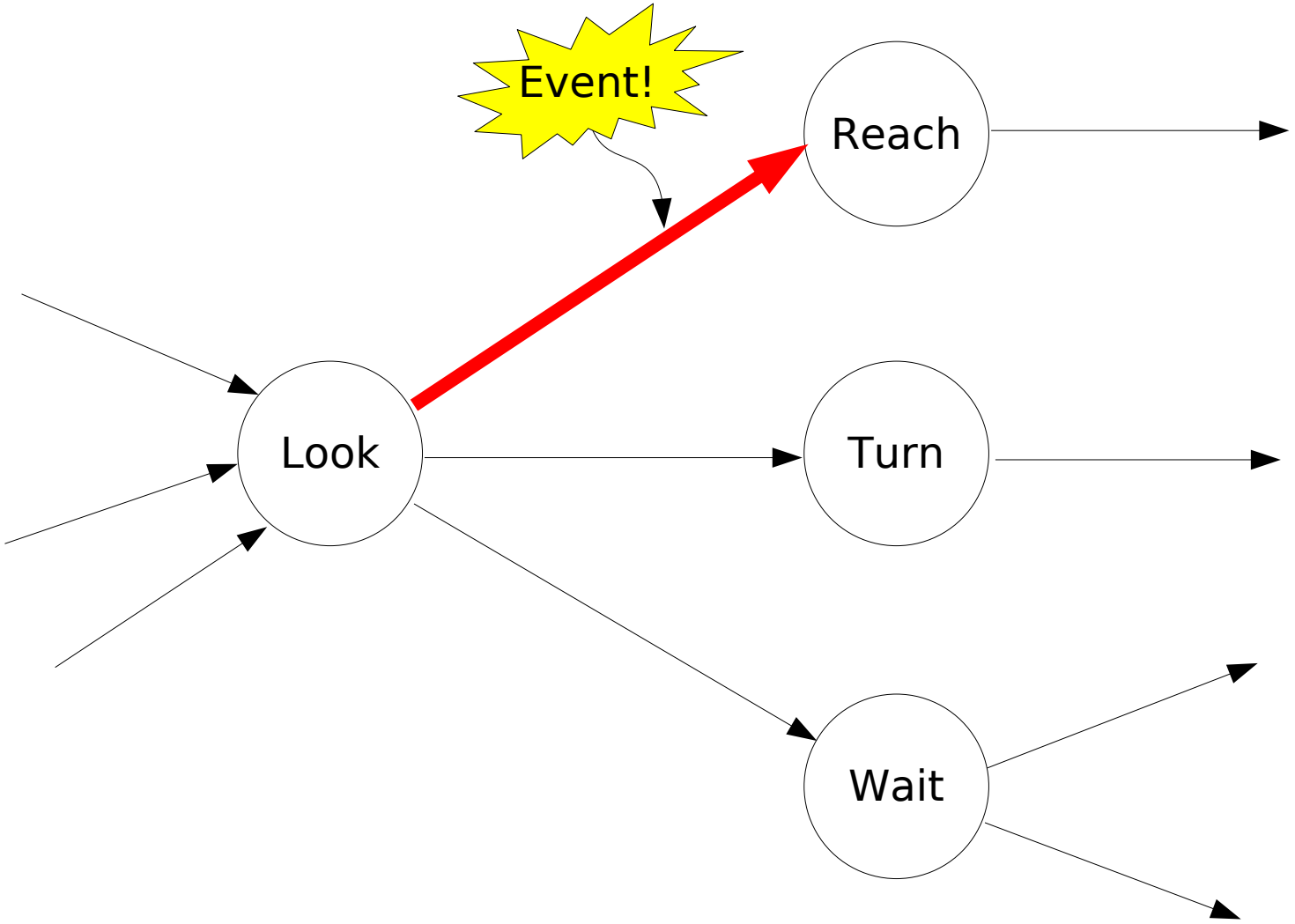
And then, something we've been looking for...



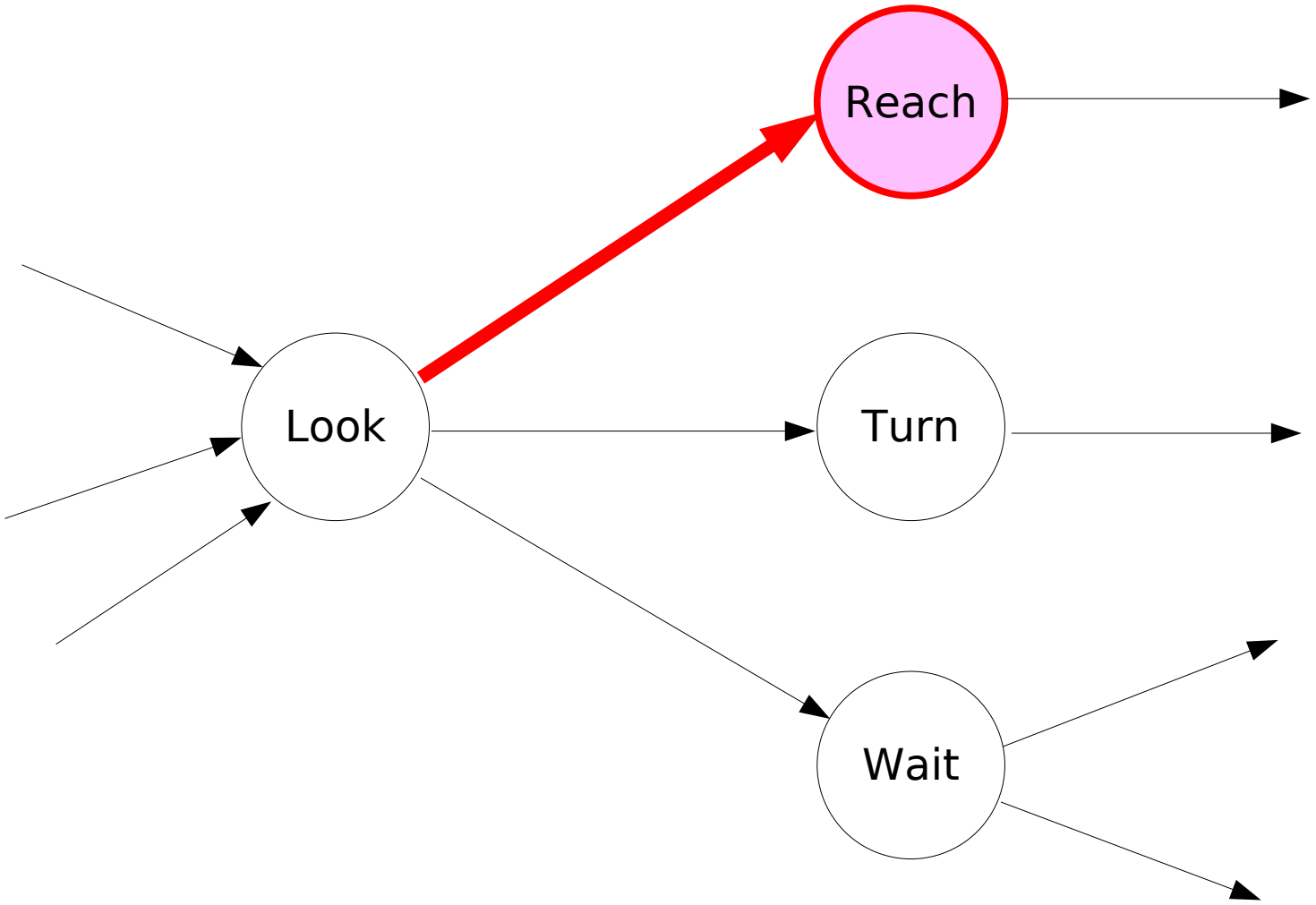
Transition decides to fire.



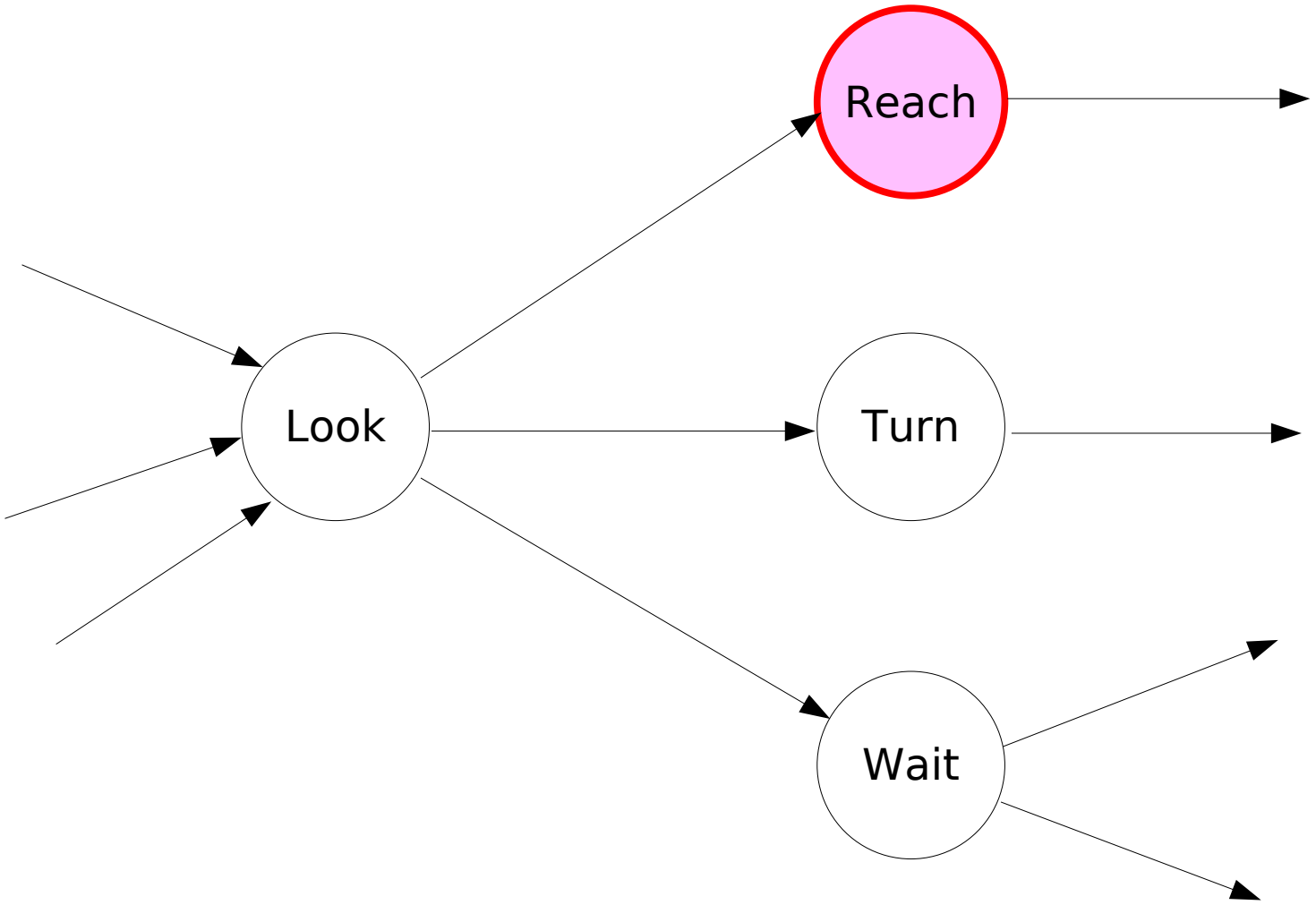
Transition deactivates the source node, Look.



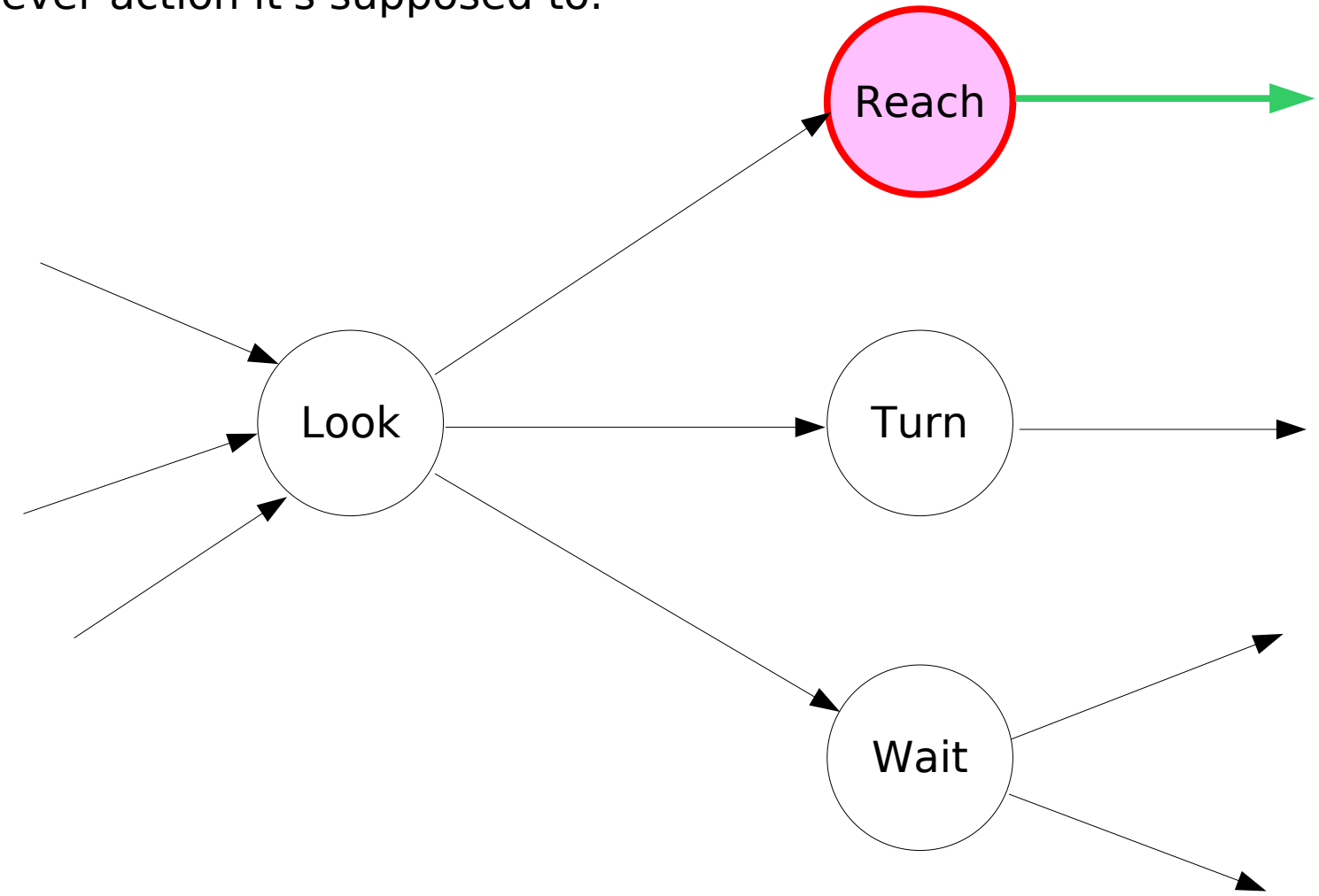
Transition activates the target node, Reach.



Transition deactivates.



Reach activates its outgoing transition, which starts listening for events as Reach performs whatever action it's supposed to.

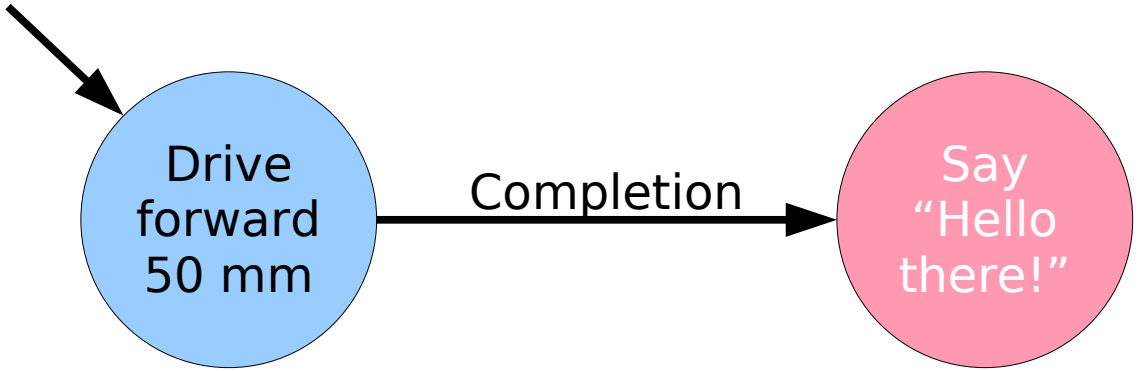


Making State Machines

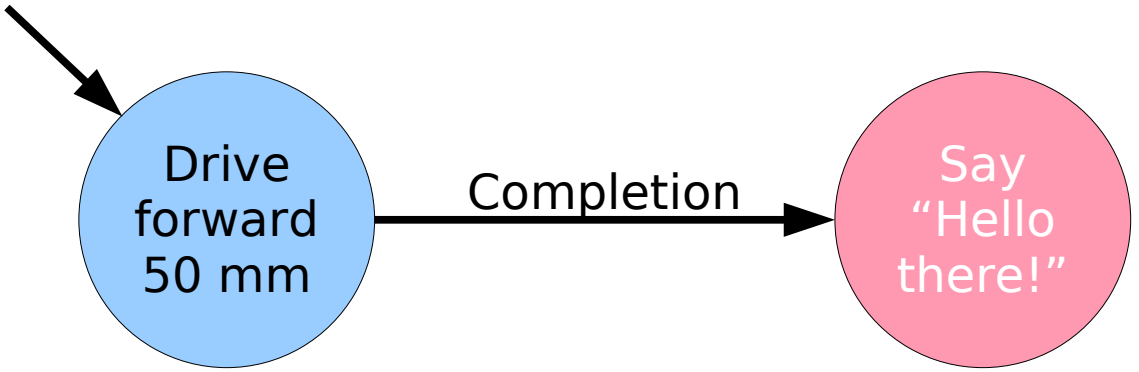
- vex-aim-tools programmers don't write Python code to build state machines one node or link at a time.
- Why not?
  - It's tedious.
  - It's error-prone.

- Instead they use a shorthand notation.
- The shorthand is turned into Python code by a state machine preprocessor, genfsm.

Example: Drive, then Talk



Example: Drive, then Talk



Shorthand notation:

Forward(50) =C=> Say(“Hello there!”)

The first defined node automatically becomes the start node.

Generated Code

```
def setup(self):

  forward1 = Forward(50)
  forward1.set_name("forward1")
  forward1.set_parent(self)

  say1 = Say('Hello there!')
  say1.set_name("say1")
  say1.set_parent(self)

  completiontrans1 = CompletionTrans()
  completiontrans1.set_name("completiontrans1")
  completiontrans1.add_sources(forward1)
  completiontrans1.add_destinations(say1)
```

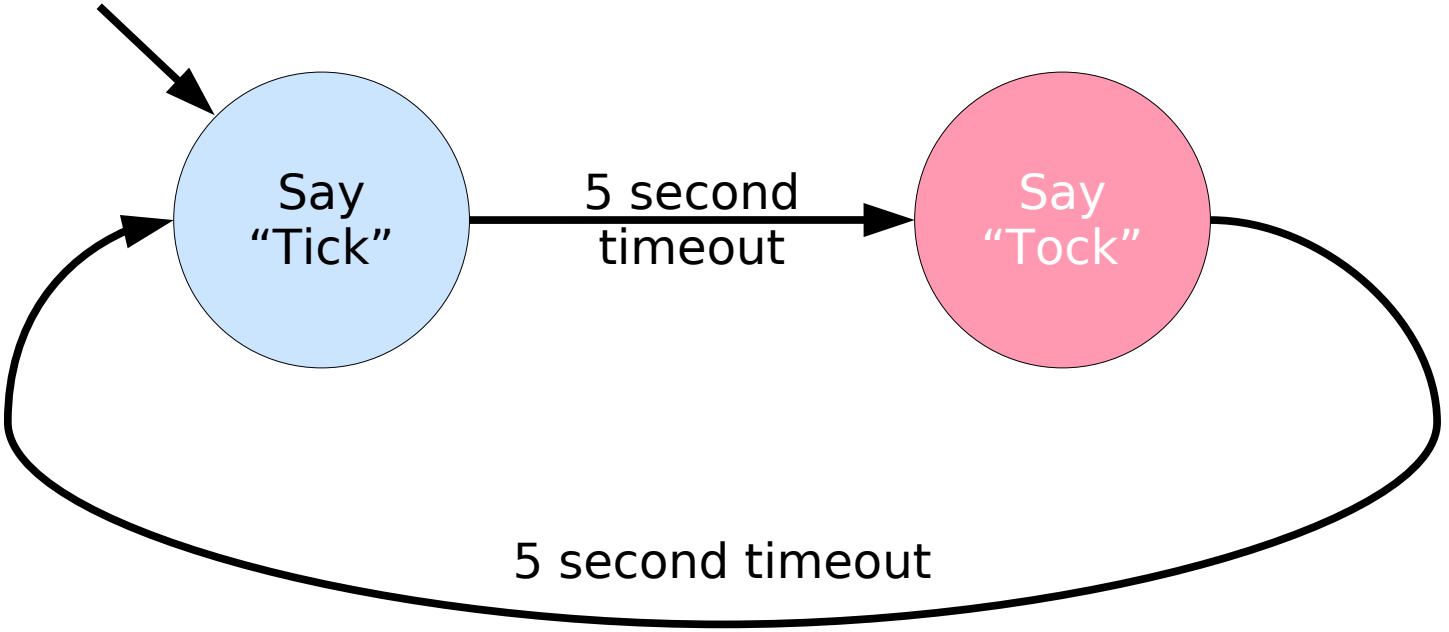
The Full Source: Example1.fsm

```
from aim_fsm import *


class Example1(StateMachineProgram):
  $setup {
    Forward(50) =C=> Say('Hello there')
  }
```

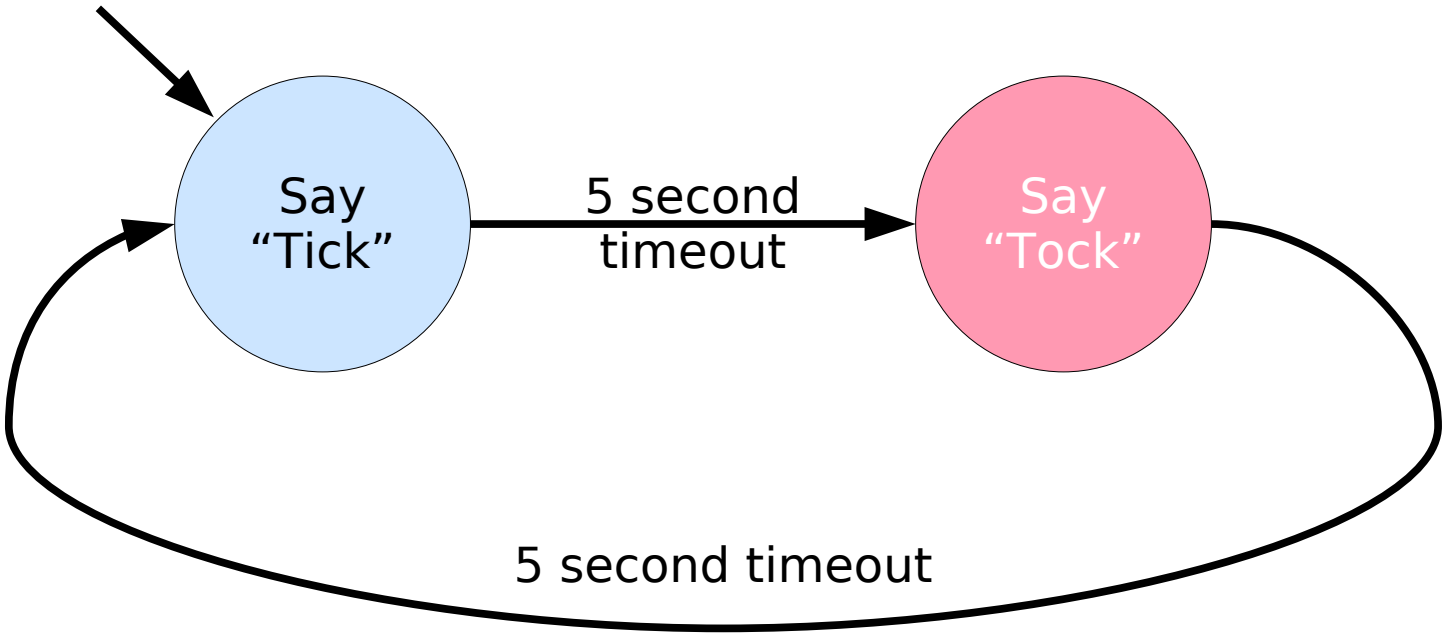
genfsm Translates .fsm to .py

```
$ genfsm Example1.fsm
Wrote generated code to Example1.py


$ simple_cli
… startup stuff …
C> runfsm('Example1')
```

Metronome



Metronome



Shorthand:

tick: Say('Tick') =T(5)=> tock

tock: Say('Tock') =T(5)=> tick

Running Nodes from the REPL

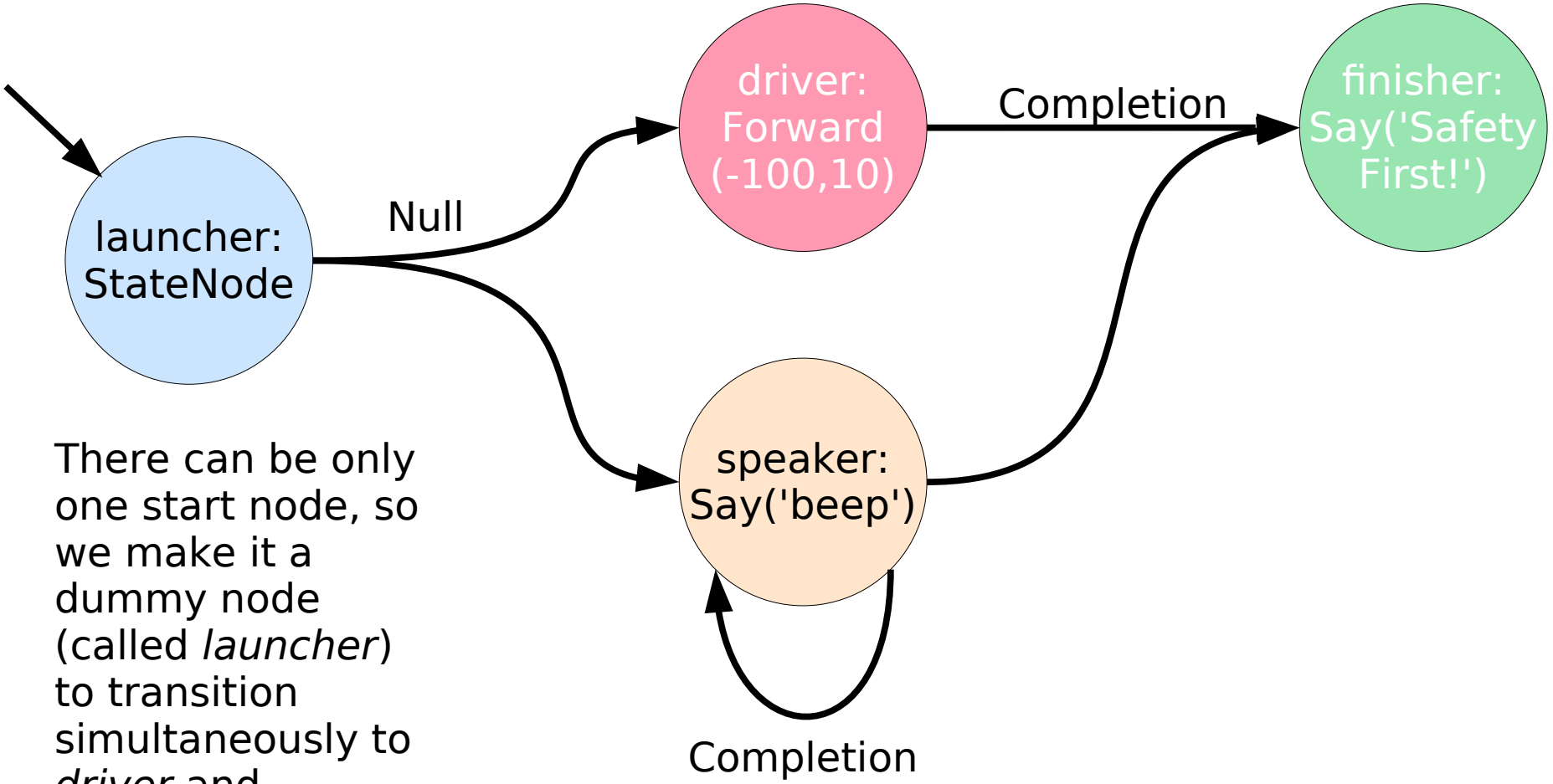
- In simple_cli, if you type Forward(50) you are calling a node constructor, not a function.
- You get back a state node object.
- It doesn't run. It's just a state node.
- Use Forward(50).now() to run it.
  - The .now() method sets up some structures the state node needs and then schedules it for immediate execution in the event loop.

Fancy State Machines

aim_fsm is a *hierarchical, parallel, message passing* state machine formalism:

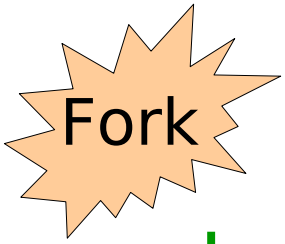
- **Hierarchical:** state machines can nest.
- **Parallel:** multiple states can be active at the same time.
- **Message passing:** transitions can transmit information to their target nodes.

“Back It Up”: Fork/Join



There can be only one start node, so we make it a dummy node (called *launcher*) to transition simultaneously to *driver* and *speaker*.

BackItUp.fsm




```
launcher: StateNode() =N=> {driver, speaker}

driver: Forward(-100, 10)
speaker: Say('Beep!') =C=> speaker
```


```
{driver, speaker} =C=>
    finisher: Say('Safety First!')
```

Defining New Node Types

- Must have a parent class that is a subclass of **StateNode**.
- Include an **__init__()** method if you need to store constructor arguments or pass values to the parent constructor.
- Include a **start()** method if you want to override the parent class's behavior.
- Include a **setup()** method (via **$setup**) if you want to include a nested state machine inside this node.

Defining New Node Types

```
class Left90(Turn):
  def __init__(self, **kwargs):
    super().__init__(angle_deg=90,
                     **kwargs)
```

Using the definition:

```
    Left90(turn_speed=30)
```

Success and Failure

```
class OrangeBarrelCheck(StateNode):
  def start(self, event=None):
    super().start(event)

    for obj in robot.world_map.objects.values():
        if isinstance(obj, OrangeBarrelObj) and
                obj.is_visible:
            self.post_success()
            return
    self.post_failure()
```

Note: a node must call super().start() before posting any events.

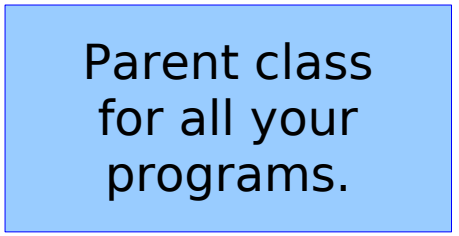
Using OrangeBarrelCheck

```
class Example2(StateMachineProgram):
  $setup {
    check: OrangeBarrelCheck()
    check =S=> Say('Visible')
    check =F=> Say('Nada')
  }
```

Constructor Arguments

```
class ObjectCheck(StateNode):
  def __init__(self, objclass):
    super().__init__()
    self.objclass = objclass


  def start(self, event=None):
    super().start(event)
    for obj in robot.world_map.objects.values():
       if isinstance(obj, self.objclass) and
                    obj.is_visible:
          self.post_success()
          return
    self.post_failure()
```

Using ObjectCheck




```
class Example3(StateMachineProgram):
  $setup {
    check: ObjectCheck(BlueBarrelObj)
    check =S=> Say('Visible')
    check =F=> Say('Nada')
  }
```

Randomness

- Say can be given a list of utterances to choose from:

```
Say(['hi', 'hello', 'howdy'])
```

- The RND transition fires immediately and chooses one destination at random.

```
launch =RND=> {eeny, meeny, miney}
```

Text Message Events

C> tm right

```
dispatch: StateNode()
dispatch =TM('forward')=> Forward(50)
dispatch =TM('right')=> Turn(-90)
```

Good Coding Style

- Node class names must begin with a capital letter.
- Node labels must be lowercase.
- It's okay to chain nodes and transitions together if each node has only one outgoing transition:

```
Forward(50) =C=>
  Say("Hi there") =C=>
    Turn(45)
```

Good Coding Style

- If a node has multiple outgoing transitions, declare the node first, then write each transition on a separate line.

```
foo: DoSomething()
foo =S=> Celebrate()
foo =F=> Mourn()
```

Determining the Start Node

The first node instance *defined* in the file is taken as the start node.

Example (terrible coding style):

```
apple =C=> pear =C=> apple

pear: Say("pear")

apple: Say("apple")
```

Don't write code like this!

The start node will be pear, not apple, since pear is the first node instance defined.

Write This Instead

```
$setup{
    apple: Say(“apple”) =C=> pear
    pear: Say(“pear”) =C=> apple
}
```

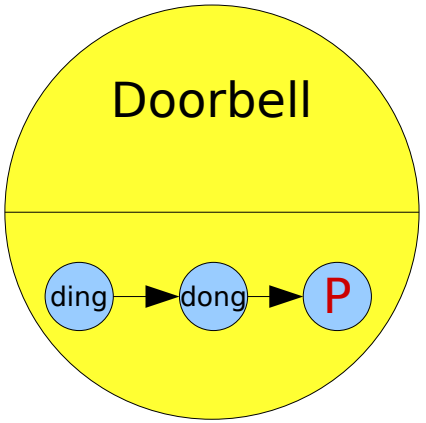
Good Coding Style

- If overriding a parent class's __init__() or start() method, be sure to:
  - call the superclass's method at the right time (this can be tricky)
  - pass arguments if appropriate.
- If overriding start() for a node that might be entered via multiple paths, be sure to check self.running and return if it’s already true, before calling the parent’s start.

Nested State Machines

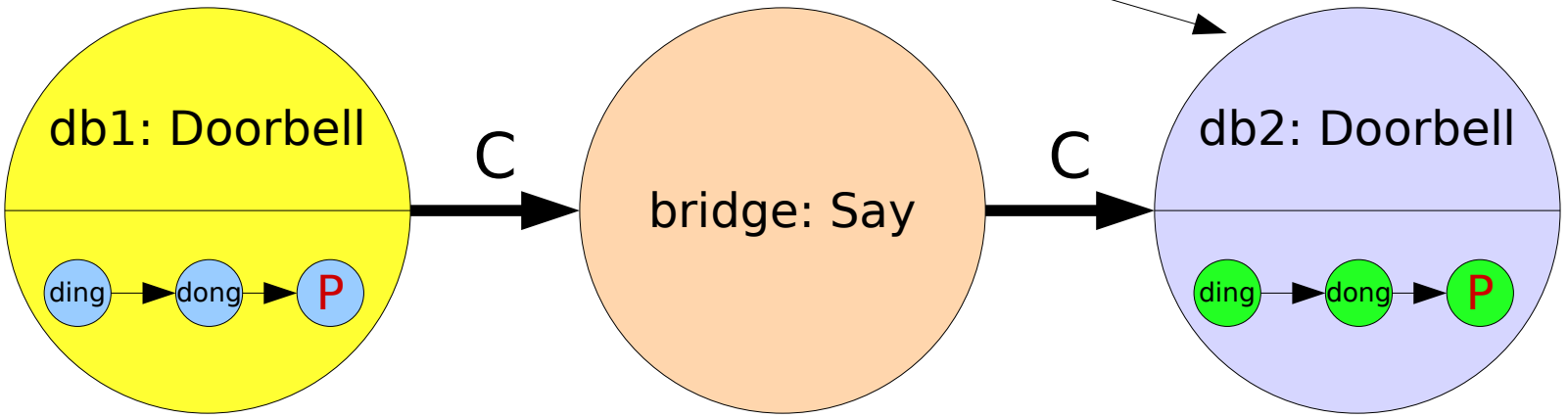
Doorbell has no start() method, but it has a setup() method.

```
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()
  }
```



Tracing

Use tracefsm(*level*) to trace execution.

0. No tracing
1. State node start
2. State node start and stop
3. Transition firing
4. Transition start and stop

5 – 9 are more advanced.

To Learn More About State Machines

- Read the aim_fsm source code.
  - See nodes.py for node types.
  - See transitions.py for transition types.

A Note About Odometry

- How does VEX AIM keep track of its position?
- Simplest method: odometry.
- Wheel encoders monitor wheel turning and accelerometers measure motion.
- Requires knowing wheel radius and encoder resolution (degrees per tick).
- Limited accuracy due to wheel slippage.
- Error is cumulative, so odometry alone is only good for the short term.
- In Lab 2 you'll test VEX AIM’s odometry.