

State Machines

15-494 Cognitive Robotics
David S. Touretzky &
Ethan Tira-Thompson

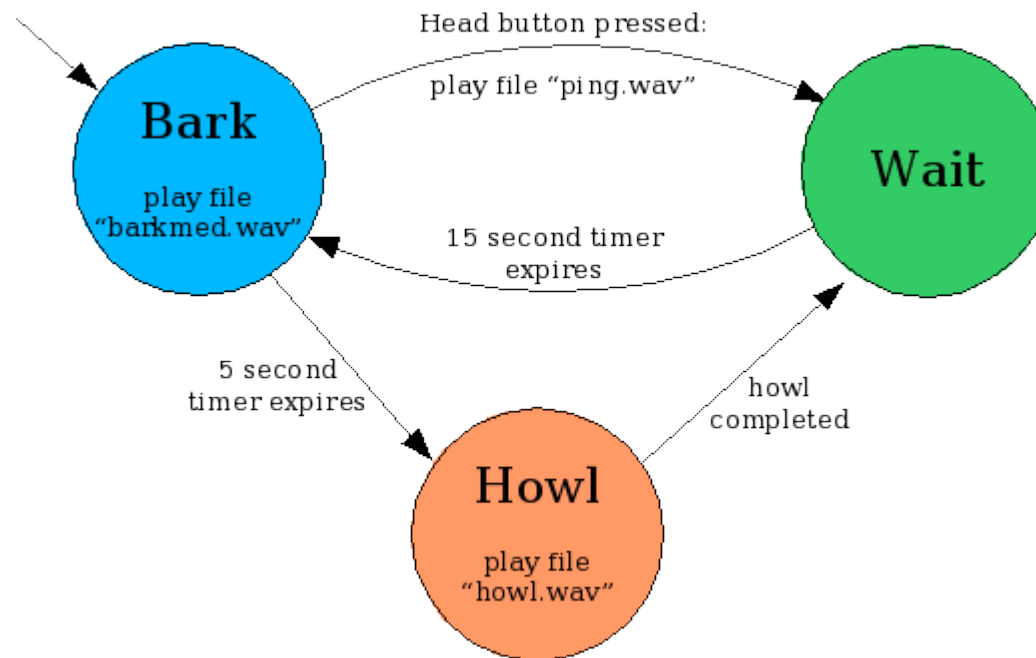
Carnegie Mellon
January 2014

Robot Control Architectures

- State machines are the simplest and most widely used robot control architecture.
- Easy to implement; easy to understand.
- Not very powerful:
 - Action sequences must be laid out in advance, as a series of state nodes.
 - No dynamic planning.
 - Failure handling must be programmed explicitly.
- But a good place to start.

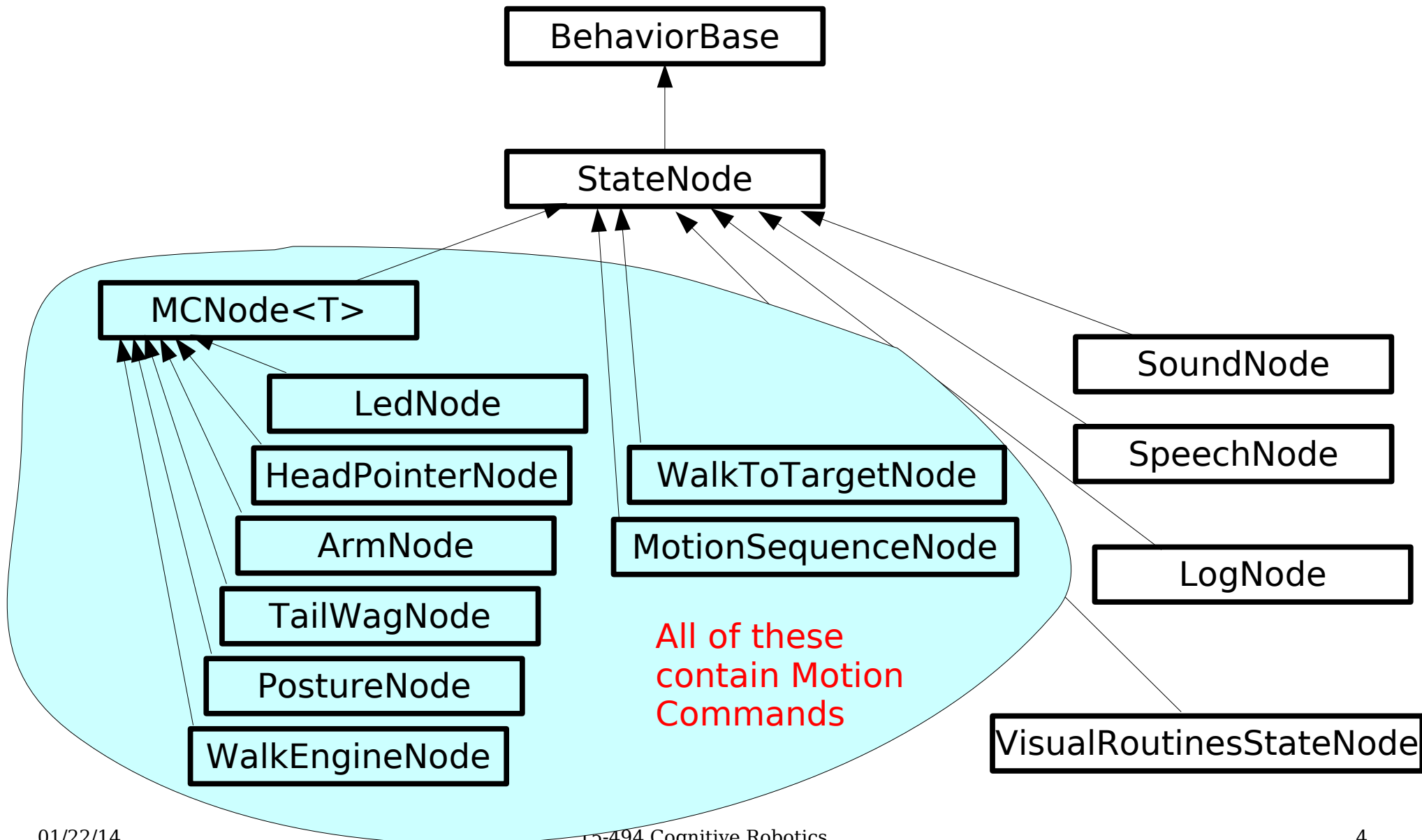
Basic Idea

- Robot moves from state to state.
- Each state has an associated action: *speak*, *move*, etc.
- Transitions triggered by sensory events or timers.

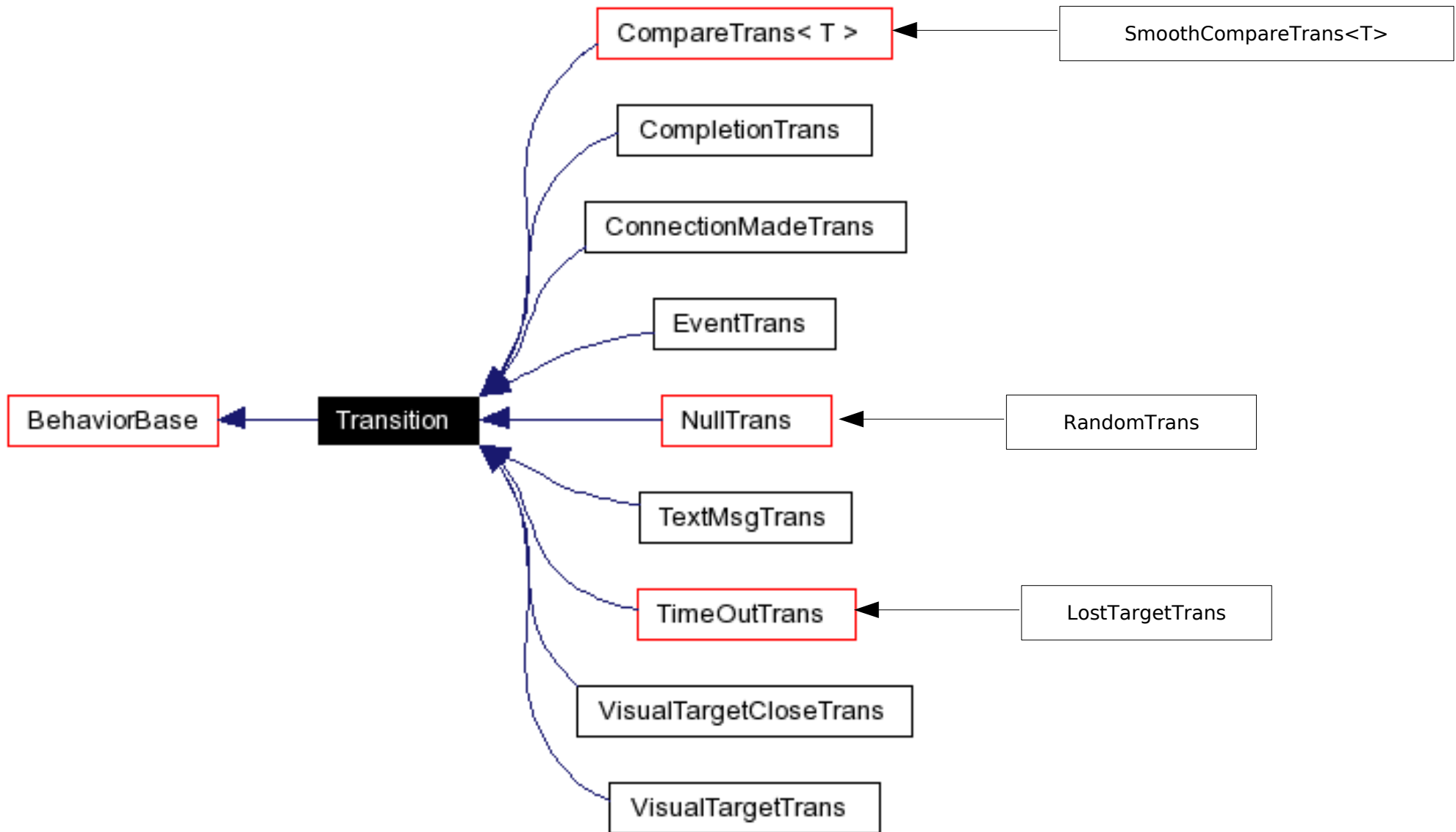


Types of State Nodes

- State nodes encapsulate complex actions, such as creating and launching a motion command.



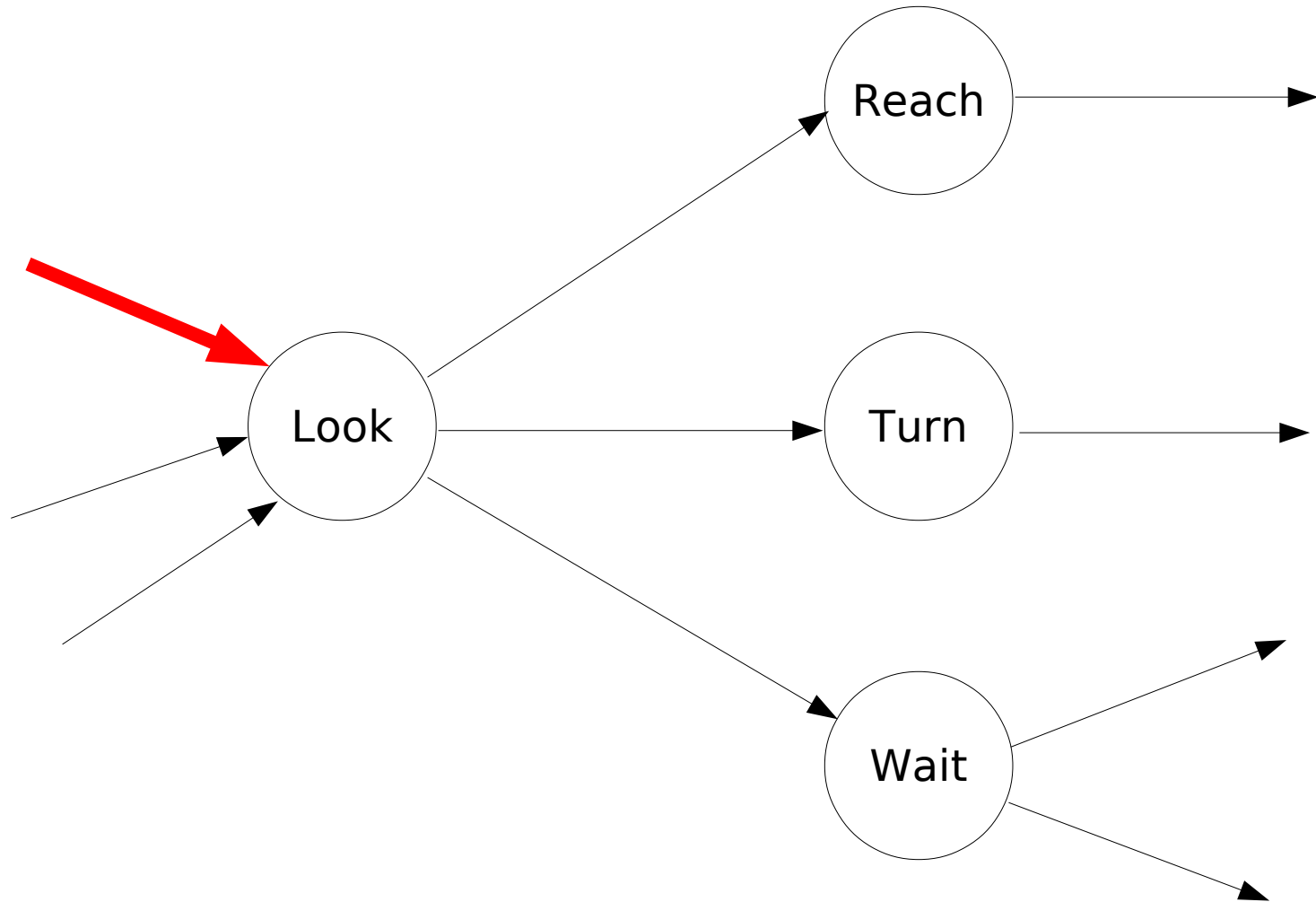
Types of Transitions



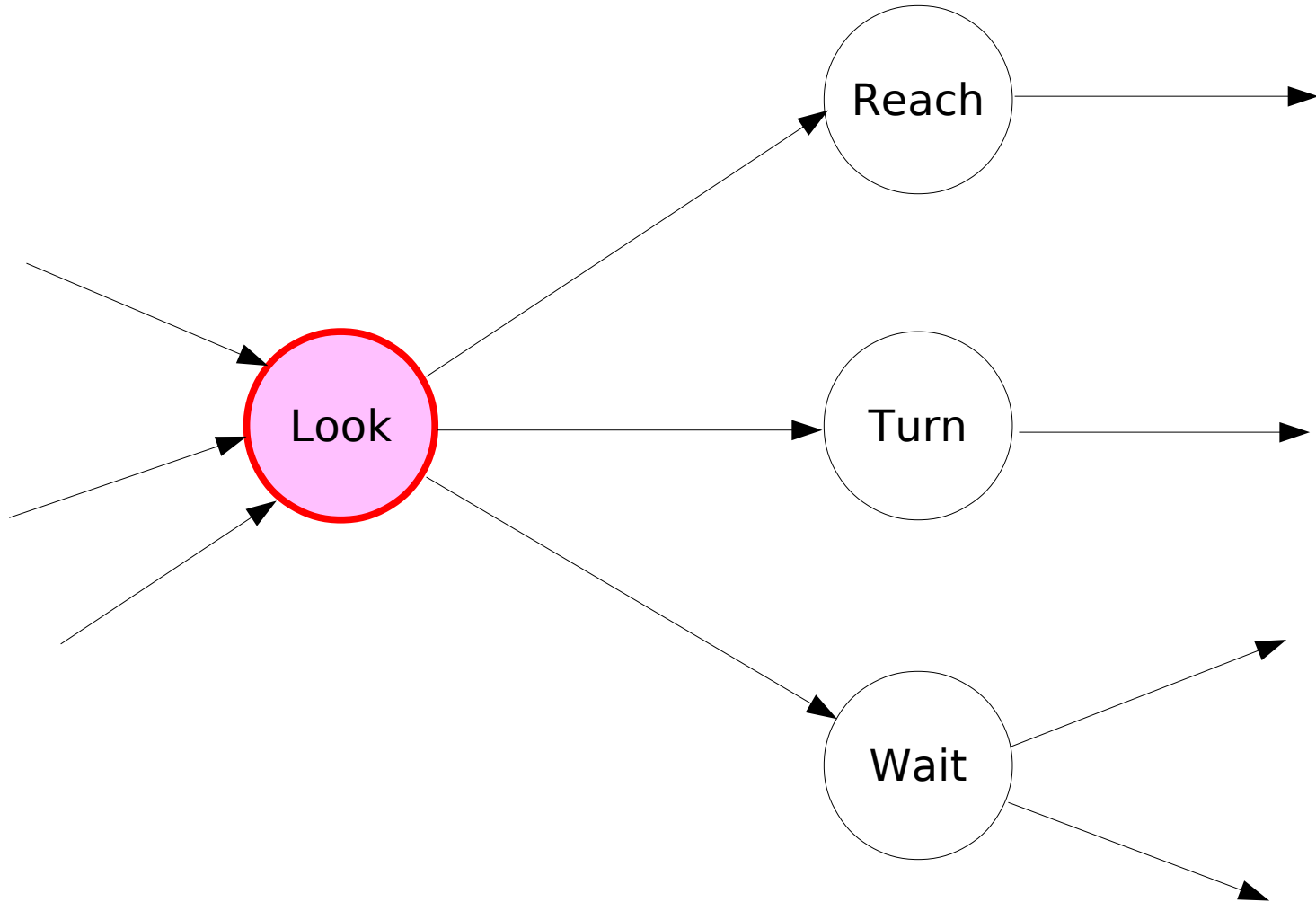
Both State Nodes and Transitions Are Behaviors

- StateNode and Transition are both subclasses of BehaviorBase.
- Tekkotsu behaviors can contain arbitrary C++ code and can generate and/or receive events.
- Transitions:
 - A transition's start() method is called whenever its *source* state node becomes active.
 - Transitions listen for sensor, timer, or other events, and when their conditions are met, they *fire*.
 - When a transition fires, it deactivates its source node(s) and then activates its target node(s).

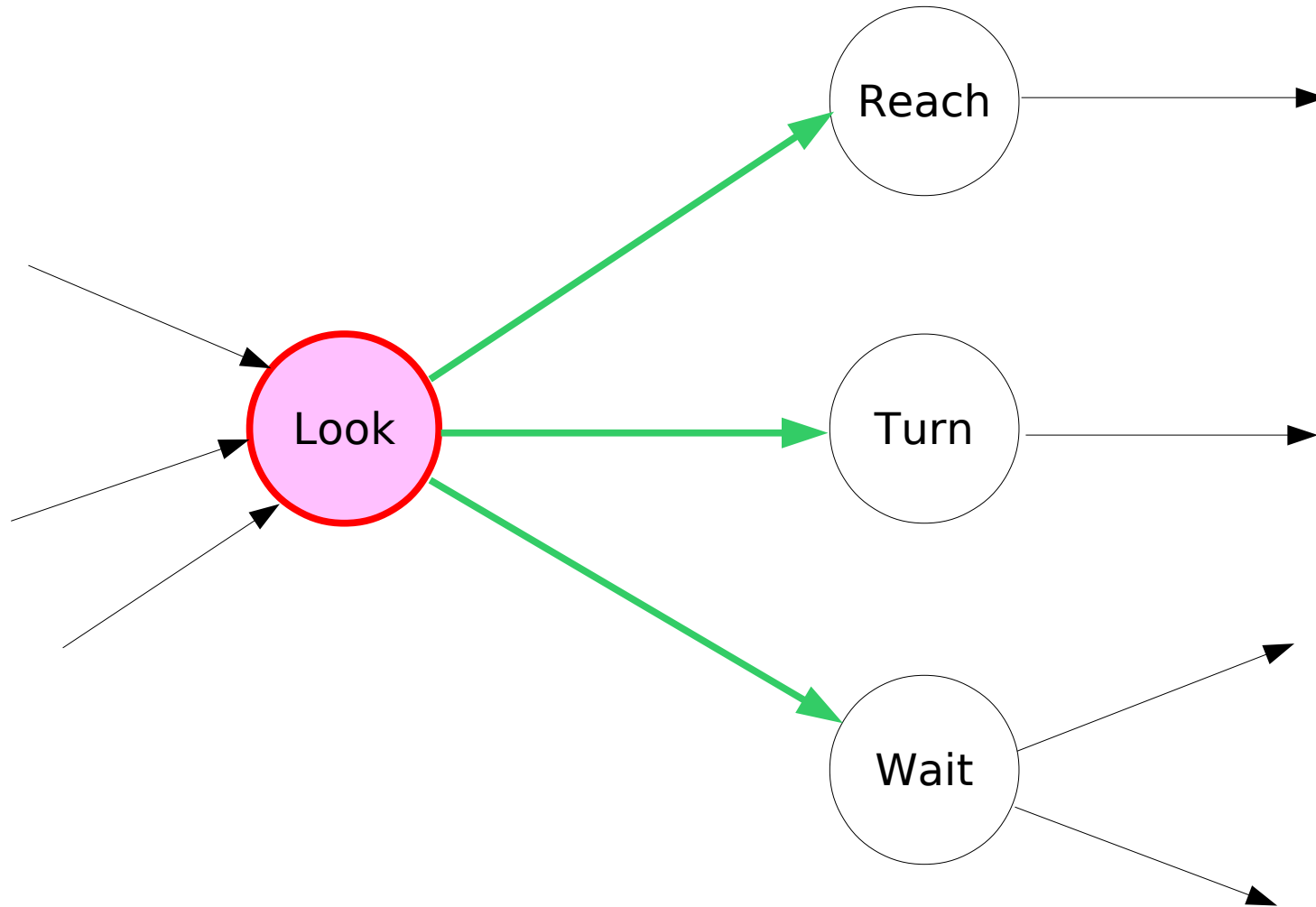
Transition firing activates state node Look.



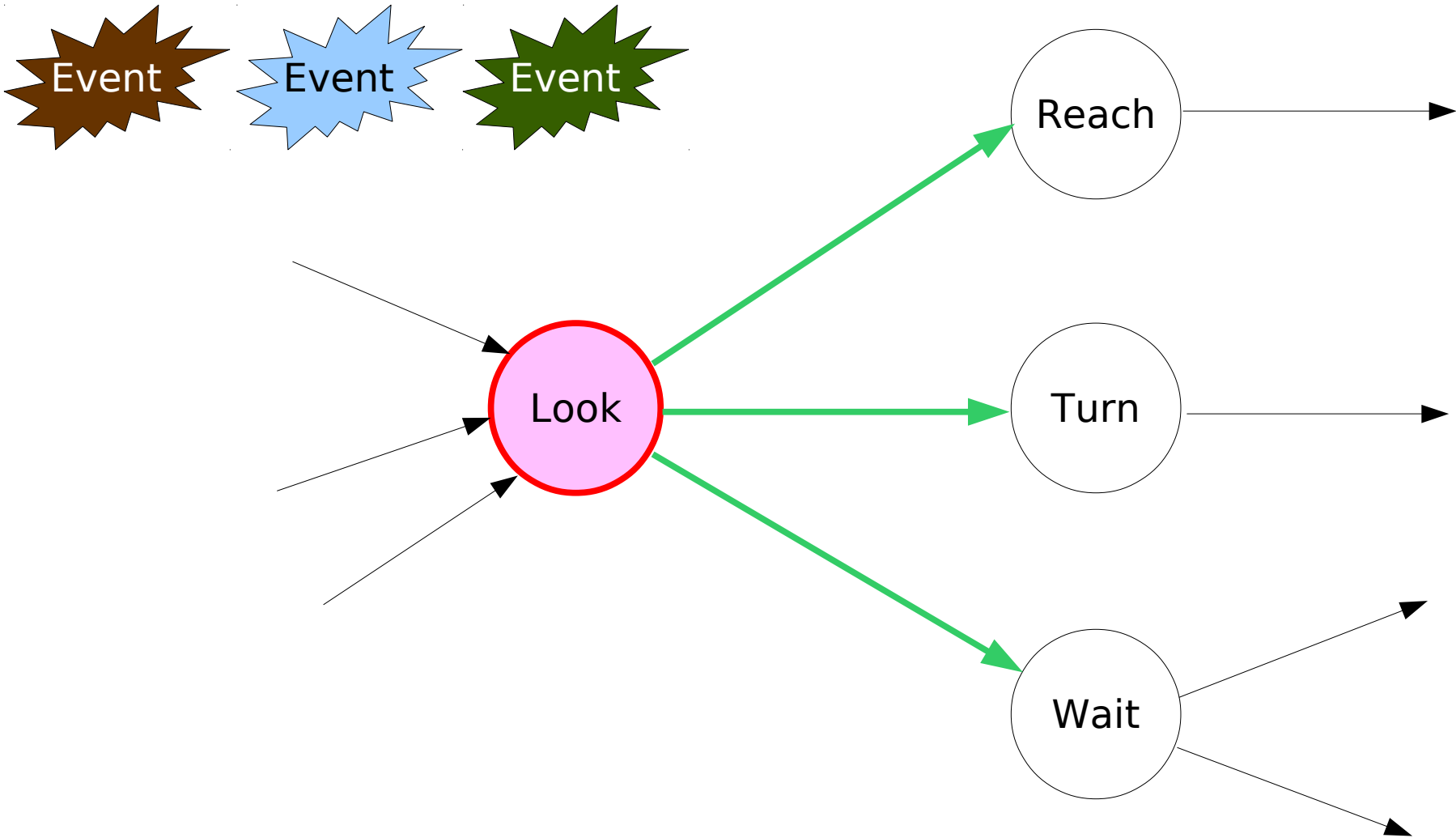
Look's start() calls StateNode::start().



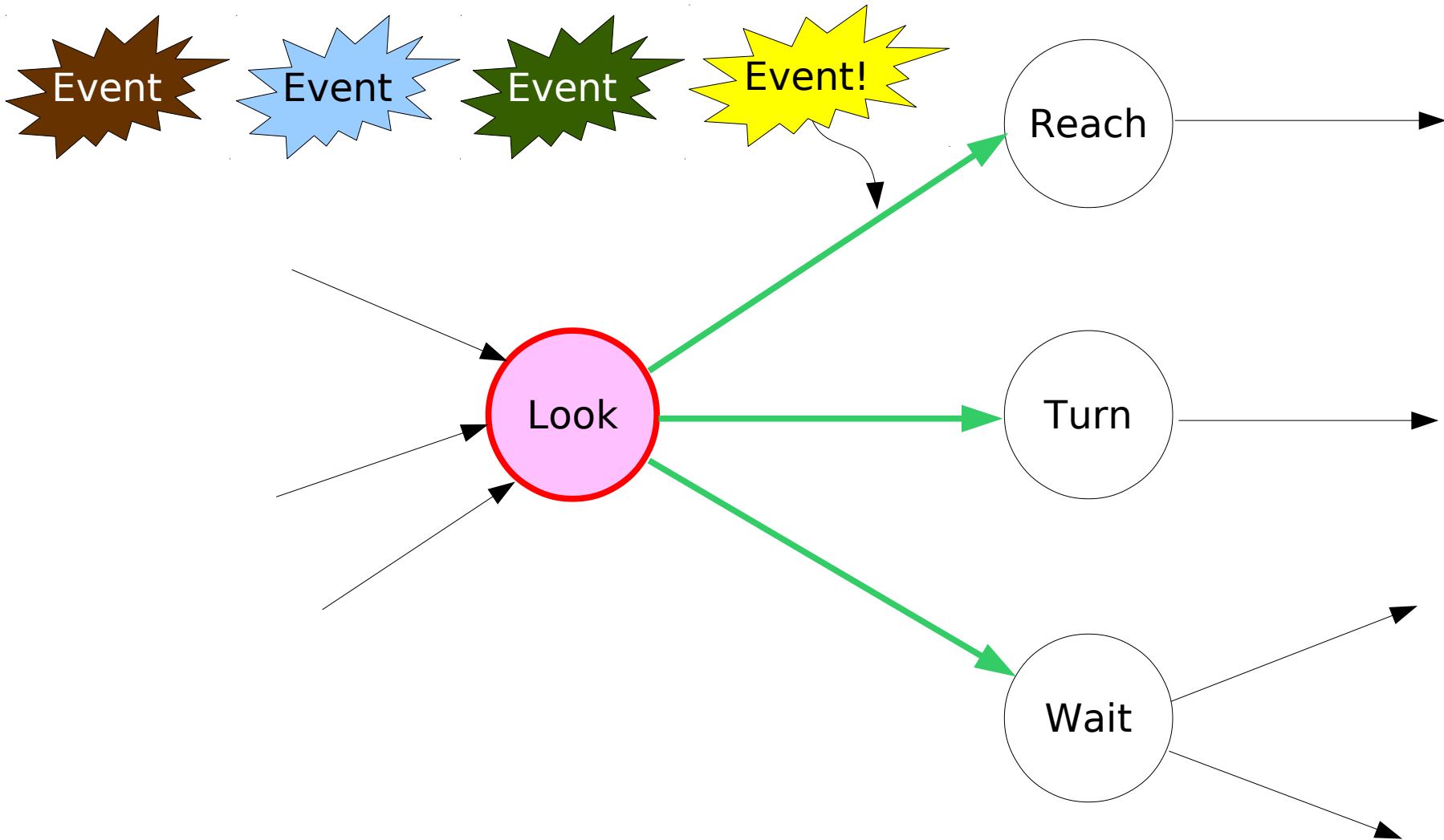
Outgoing transitions become active and begin listening for events.



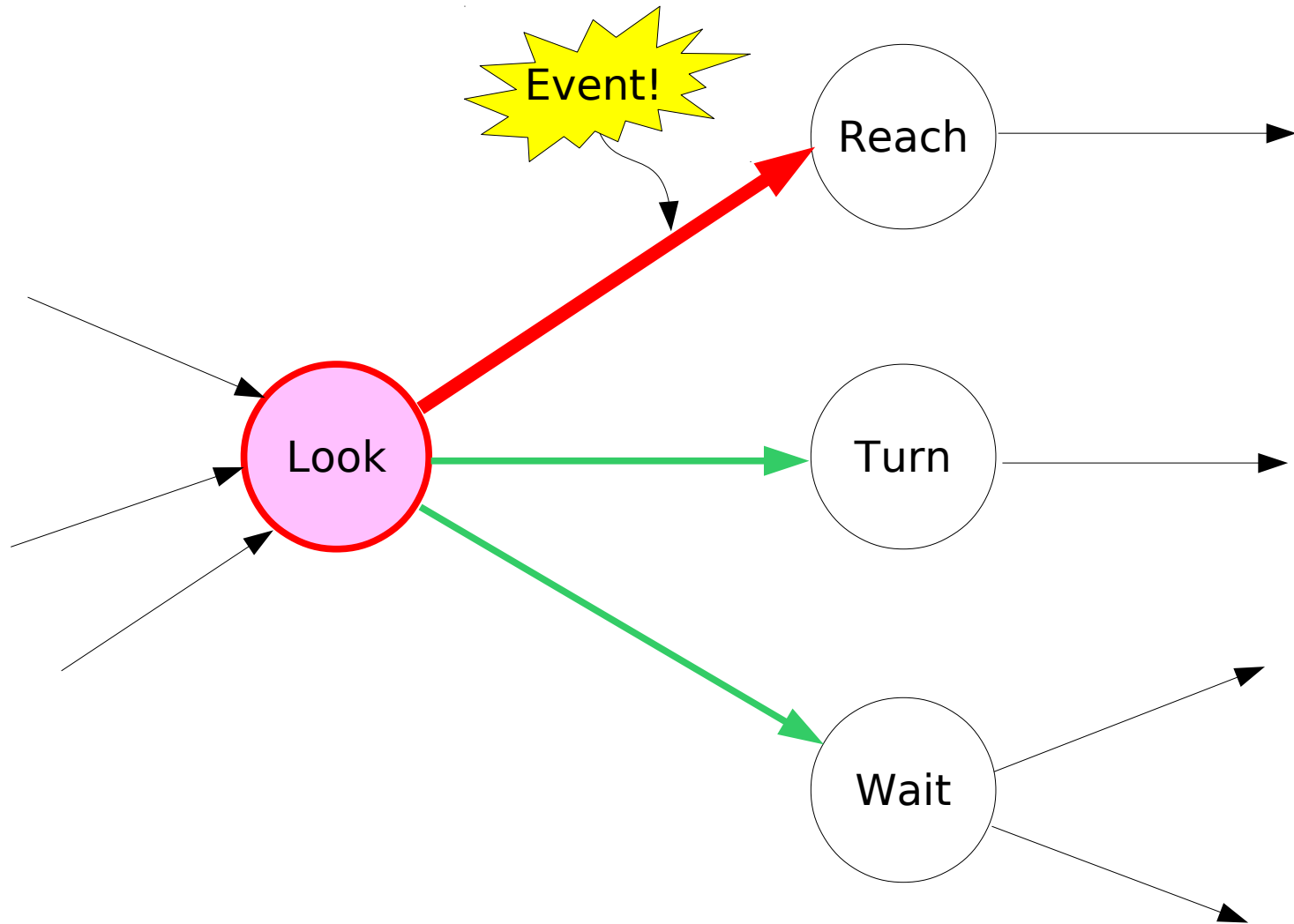
Random things happen....



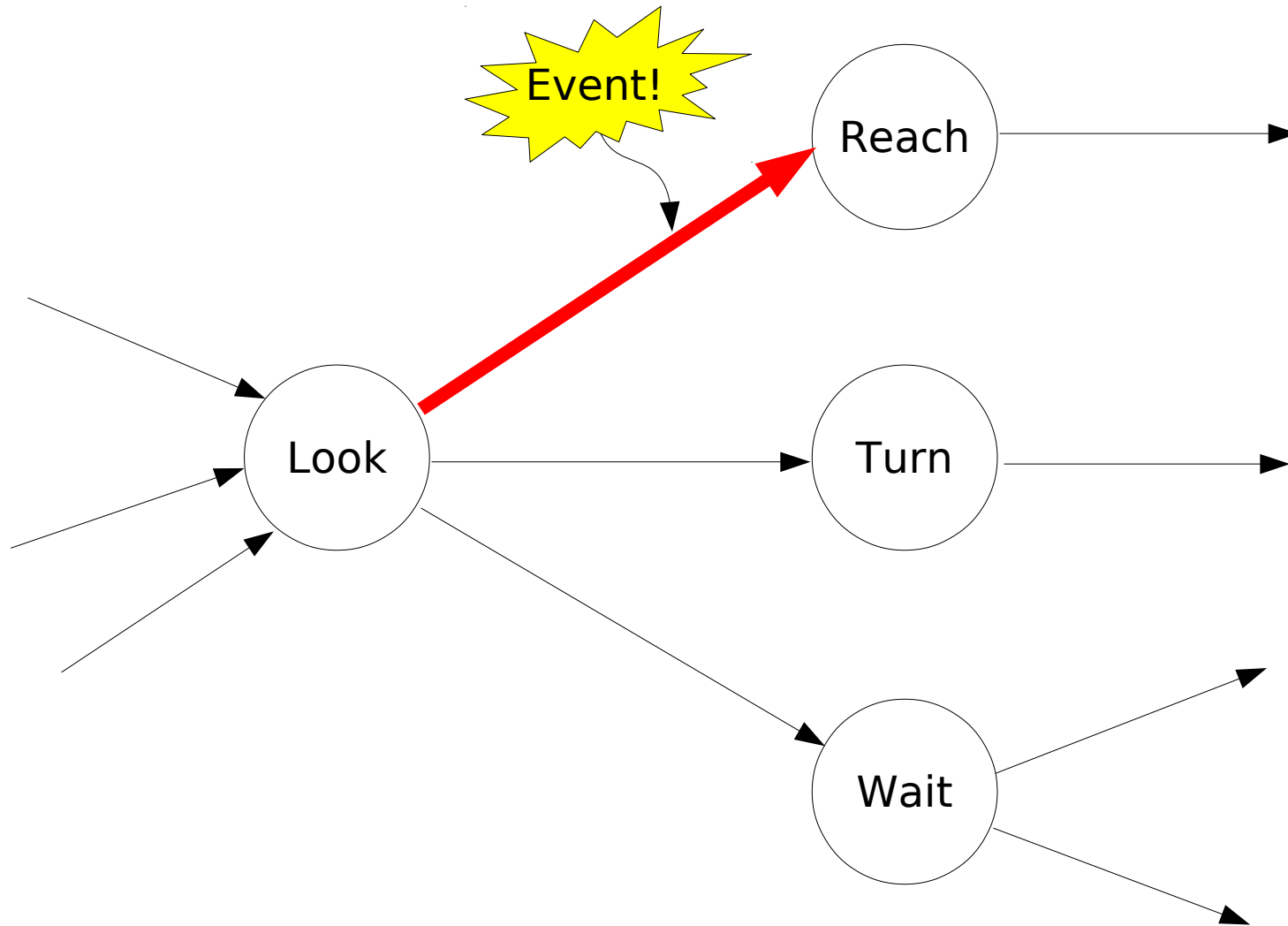
And then, something we've been looking for...



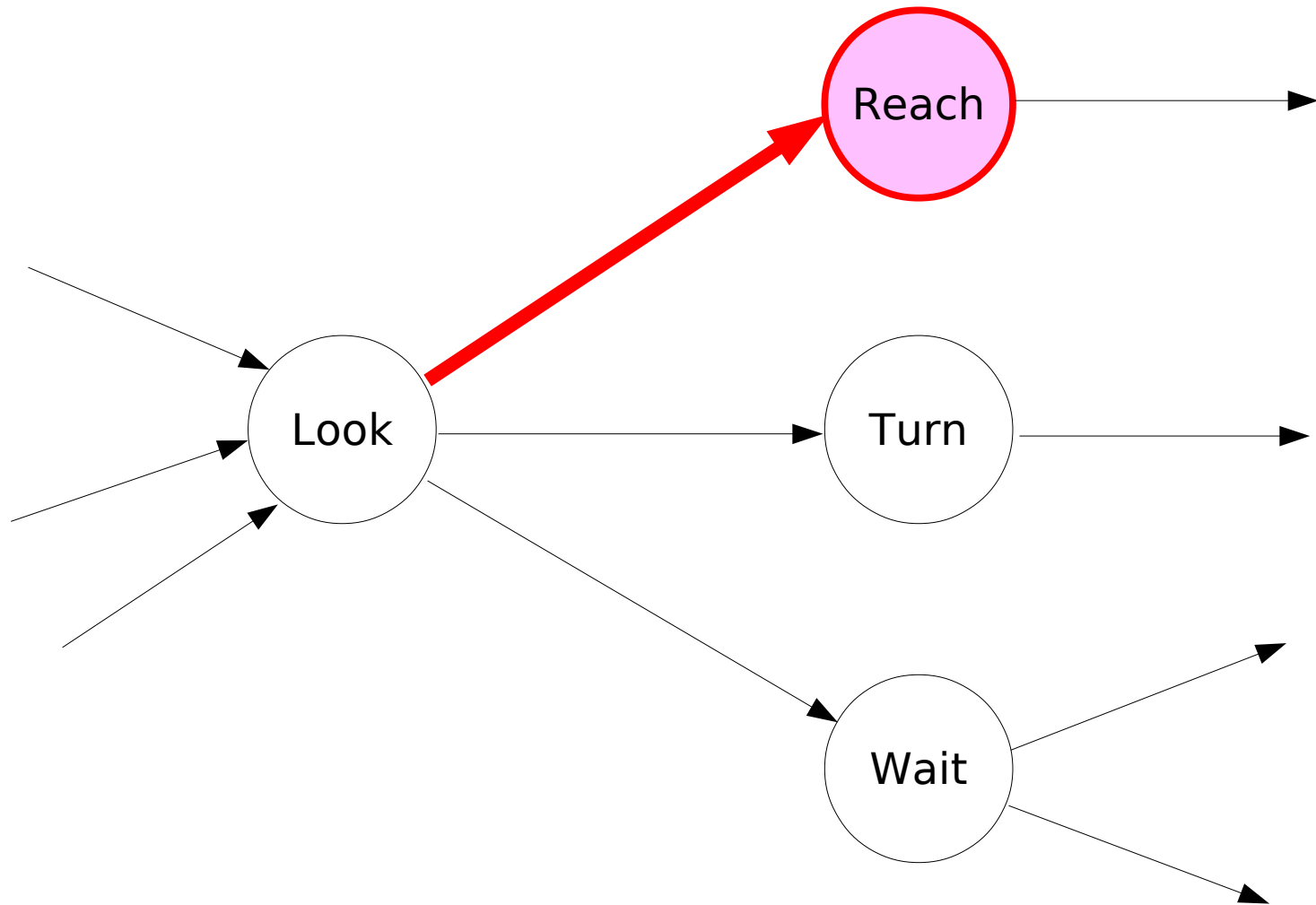
Transition decides to fire.



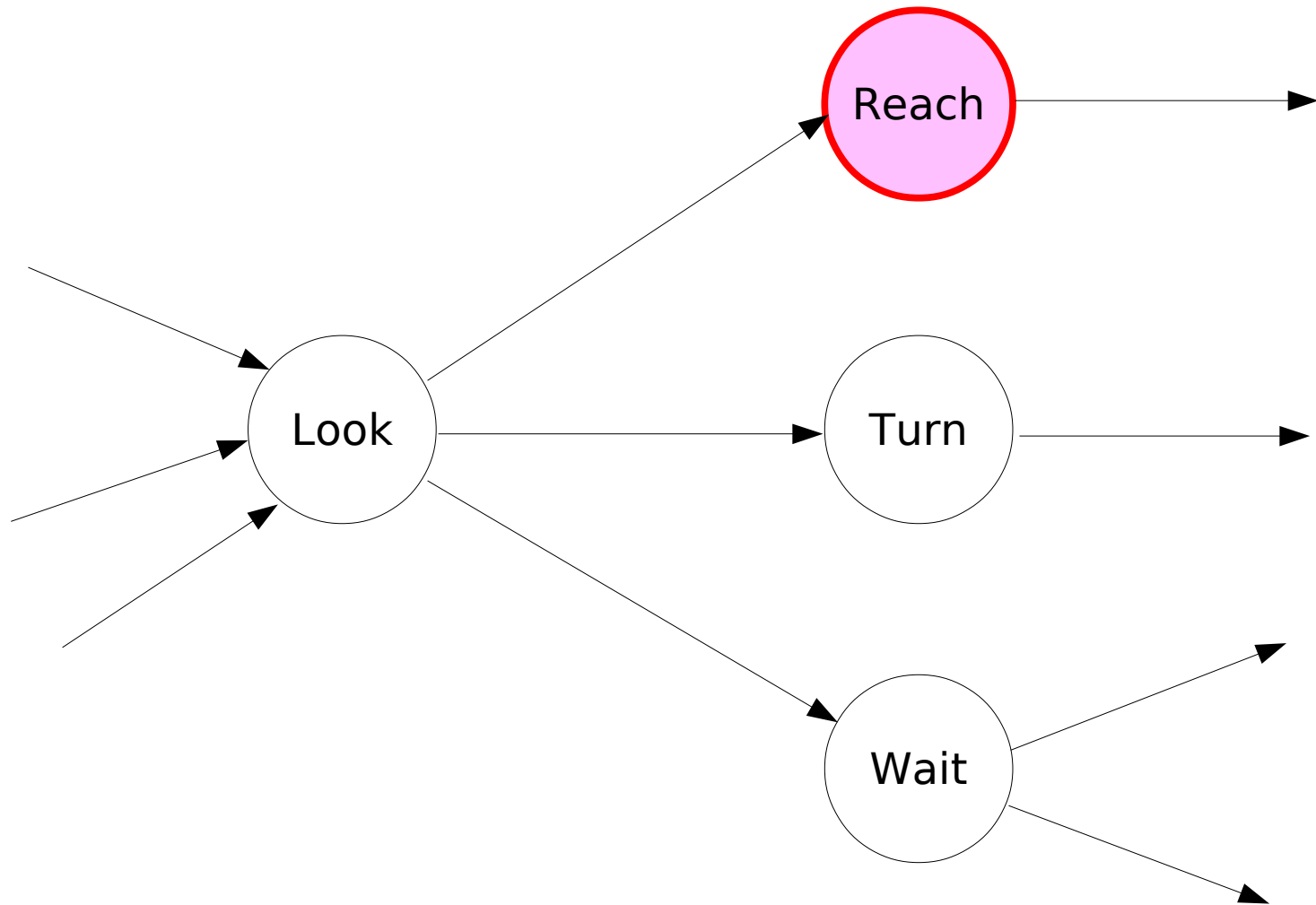
Transition deactivates the source node, Look.



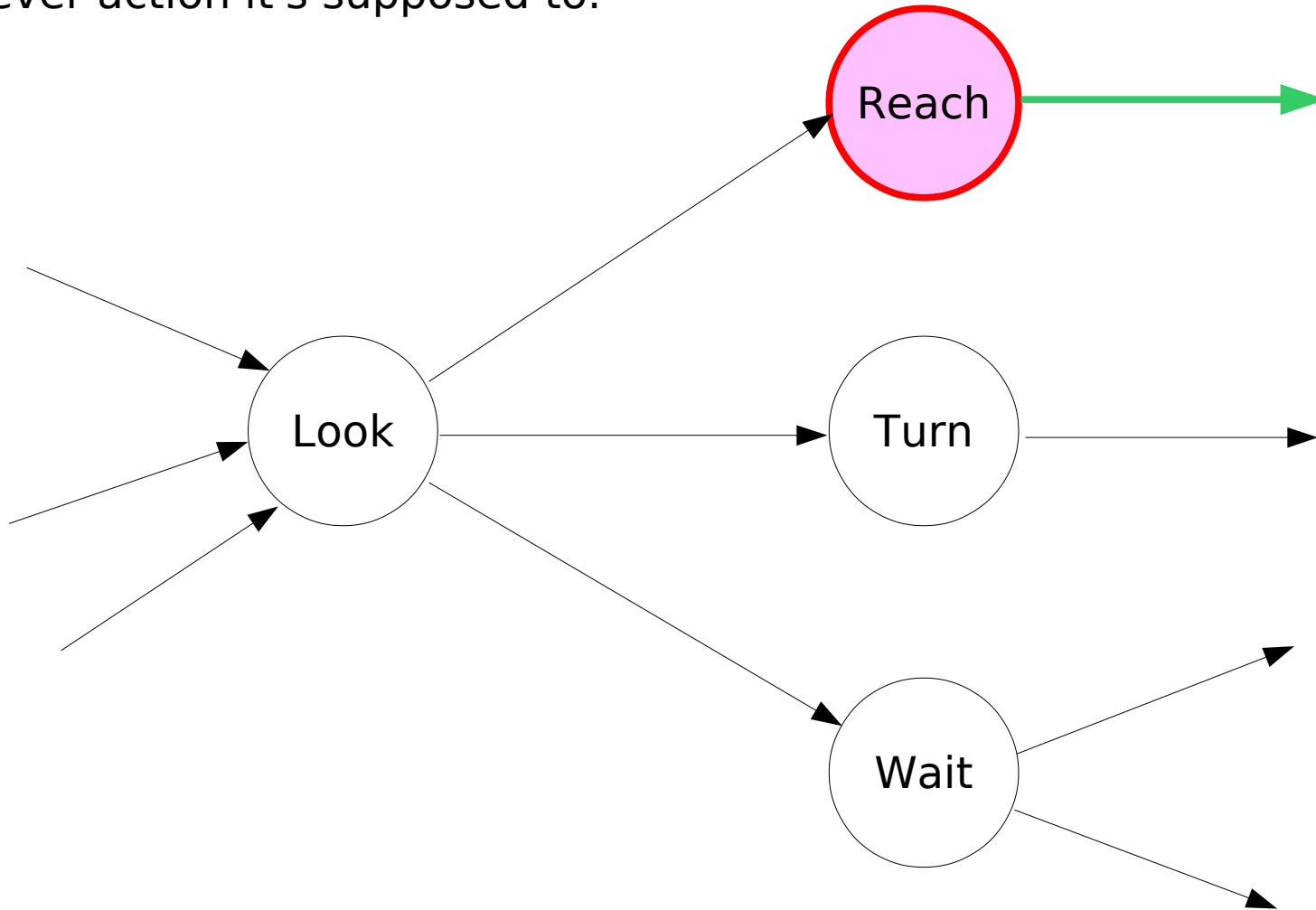
Transition activates the target node, Reach.



Transition deactivates.



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



State Machine Compiler

- Tekkotsu programmers don't normally write C++ code to build state machines one node or link at a time.

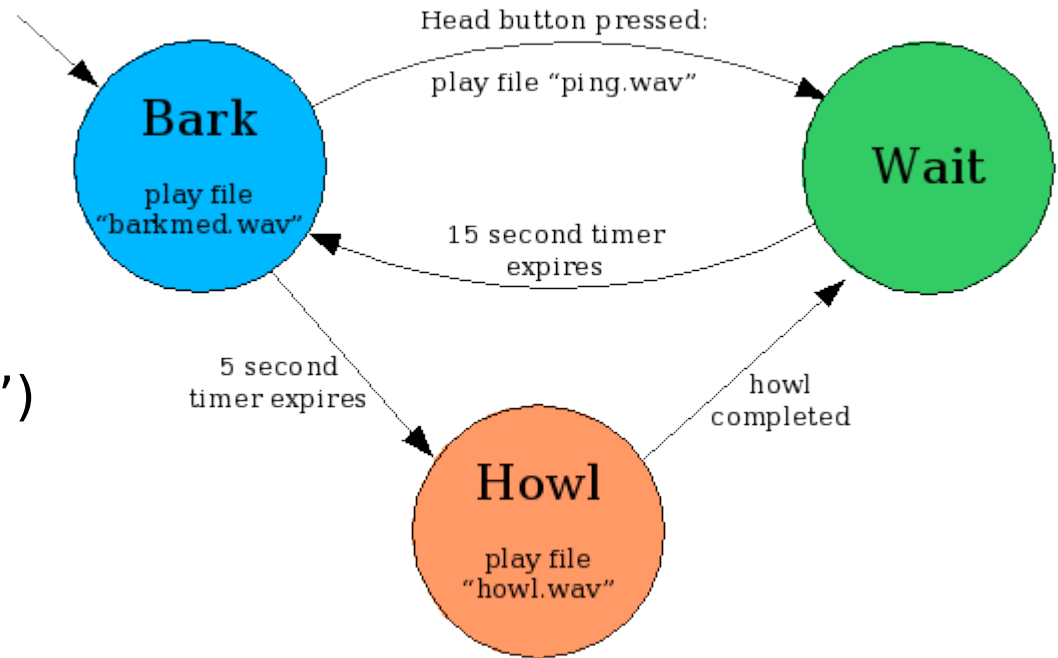
- Why not?

- It's tedious.
- It's error-prone.



- Instead they use a shorthand notation.
- The shorthand is turned into C++ by a state machine compiler.

Shorthand Notation



bark: SoundNode("barkmed.wav")

howl: SoundNode("howl.wav")

wait: StateNode

bark =T(5000)=> howl

bark =B(RobotInfo::PlayButOffset)[setSound("ping.wav")]=> wait

howl =C=> wait

wait =T(15000)=> bark

Real Code: AnnoyingDog.cc.fsm

```
#include "Behaviors/StateMachine.h"
```

```
$nodeclass AnnoyingDog : StateNode {
```

```
  $setupmachine{
```

```
    bark: SoundNode("barkmed.wav")
```

```
    howl: SoundNode("howl.wav")
```

```
    wait: StateNode
```

```
    bark =T(5000)=> howl
```

```
    bark =B(RobotInfo::PlayButOffset)[setSound("ping.wav")]=> wait
```

```
    howl =C=> wait
```

```
    wait =T(15000)=> bark
```

```
  }
```

```
}
```

```
REGISTER_BEHAVIOR(AnnoyingDog);
```

Advanced Shorthand: Chaining

- “Kiddie code”:

```
say_hi: SpeechNode("Hi")  
say_bye: SpeechNode("Bye")  
say_why: SpeechNode("Why")
```

```
say_hi =T(3000)=> say_bye
```

```
say_bye =T(3000)=> say_why
```

- Chained code:

```
SpeechNode("hi") =T(3000)=>  
  SpeechNode("bye") =T(3000)=>  
    SpeechNode("why")
```

Good Coding Style

- If a node has multiple outgoing transitions, don't use chaining.
 - Define the node first, on a separate line, with a label.
 - Then write each of the transitions below it.
- It's good to chain if a node has only one transition.
- Example:

```
look: LookForToys
```

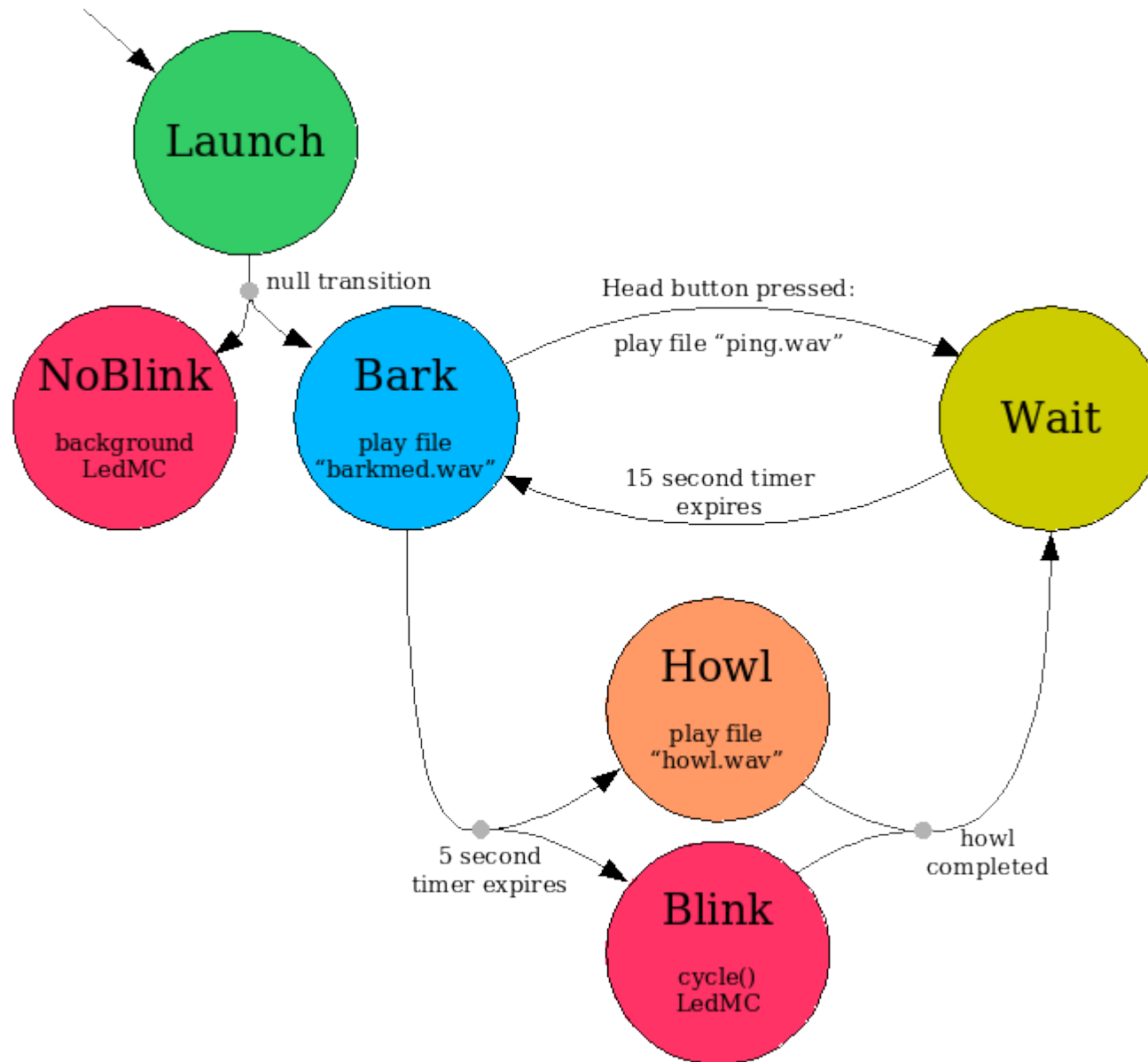
```
look =S=> SpeechNode("a toy!") =C=> trygrab
```

```
look =F=> askforhelp
```

Extensions to the Basic Formalism

- Extension 1: multi-states (parallelism).
 - Several states can be active at once.
 - Provides for parallel processing (but coroutines, not threads).
- Extension 2: hierarchical structure.
 - State machines can nest inside other state machines.
- Extension 3: message passing.
 - When a state posts an event that triggers a transition, it can include a message that will be passed to the destination state.
 - This makes state transitions resemble procedure calls.

Multi-State Machines



Blink Using LedEngine::cycle()

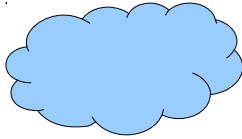
- Blink uses a motion command called LedMC, which is a child of LedEngine.
- The LedEngine::cycle() method never completes.
- When the howl completes, we want to leave both the howl state and the blink state.
- We can do this by telling CompletionTrans that only one of its source nodes needs to signal a completion in order for the transition to fire.
- When it does fire, it will deactivate both source nodes.


```
$setupmachine{
```

```
// Annoying dog with blinking LEDs
```

```
launch: StateNode =N=> {noblink, bark}
```

```
noblink:
```



```
bark: SoundNode("barkmed.wav")
```

```
bark =B(PlayButOffset)[setSound("ping.wav")]=> wait
```

```
bark =T(5000)=> {howl, blink}
```

```
howl: SoundNode("howl.wav")
```

```
blink: LedNode[getMC()->cycle(RobotInfo::AllLEDMask, 1500, 1.0)]
```

```
{howl, blink} =C(1)=> wait
```

```
wait: StateNode =T(15000)=> bark
```

```
}
```

What if we instead wrote this?
`{howl, blink} =C=> wait`

NoBlink in the Background

- When the robot isn't howling, we want all its LEDs to stay dark.
- But we can terminate the Blink node at any time; the LedNode might leave the LEDs in a partially-on state.
- Solution: have a second LEDNode called NoBlink programmed to keep the LEDs dark, but assign it a low priority.
- The Blink node will override NoBlink when it's active.
- When Blink is not active, NoBlink will keep the LEDs dark.

```
$setupmachine{  
  // Annoying dog with blinking LEDs  
  
  launch: StateNode =N=> {noblink, bark}  
  
  noblink: LedNode [setPriority(MotionManager::kBackgroundPriority);  
                   getMC()->set(RobotInfo::AllLEDMask,0.0)]  
  
  bark: SoundNode("barkmed.wav")  
  bark =B(PlayButOffset)[setSound("ping.wav")]=> wait  
  bark =T(5000)=> {howl, blink}  
  
  howl: SoundNode("howl.wav")  
  
  blink: LedNode[getMC()->cycle(RobotInfo::AllLEDMask, 1500, 1.0)]  
  
  {howl, blink} =C(1)=> wait  
  
  wait: StateNode =T(15000)=> bark  
  
}
```

Summary of Shorthand Notation

- Instantiating a node:

label: NodeClass(constructor_args)[initializers]

Labels must begin with a lowercase letter.

Class names must begin with an uppercase letter.

- Transition, short form examples:

source =C=> target

source =T(n)=> target

source =E(g,s,t)=> target

- Transition, long form:

source >== transname:

TransitionClass(constructor_args)[initializers] ==> targetnode

- Multiple sources/targets:

{src1, src2, ...} =Transition=> {targ1, targ2, ...}

Short and Long Forms

<code>>==NullTrans==></code>	<code>=N=></code>
<code>>==CompletionTrans==></code>	<code>=C=></code>
<code>>==CompletionTrans(n)==></code>	<code>=C(n)==></code>
<code>>==TimeoutTrans(t)==></code>	<code>=T(t)==></code>
<code>>==EventTrans(g,s,t)==></code>	<code>=E(g,s,t)==></code>
<code>>== EventTrans(EventBase::buttonEGID, s) ==></code>	<code>=B(s)==></code>
<code>>== TextMsgTrans(str)==></code>	<code>=TM(str)==></code>
<code>>==RandomTrans==></code>	<code>=RND=></code>
<code>>==SignalTrans<T>==></code>	<code>=S<T>==></code>
<code>>==SignalTrans<T>(v)==></code>	<code>=S<T>(v)==></code>
<code>success or failure transitions</code>	<code>=S=> or =F=></code>

Defining the Start Node

- If there is a node labeled **startnode**, it will be taken as the start node of the state machine.
- If there is no startnode, then the first node instance *defined* in the file is taken as the start node.
- Example:

```
apple =C=> pear =C=> apple  
pear: SpeechNode("pear")  
apple: SpeechNode("apple")
```

The start node will be pear, since it is the first node instance defined.

Defining New Node Classes

```
#include "Behaviors/StateMachine.h"

$class MyMachine : StateNode {

    $class Greet : StateNode : doStart {
        cout << "Hello there!" << endl;
    }

    $class Sendoff : StateNode : doStart {
        cout << "So long!" << endl;
    }

    $setupmachine{
        startnode: Greet =T(5000)=> Sendoff
    }

}

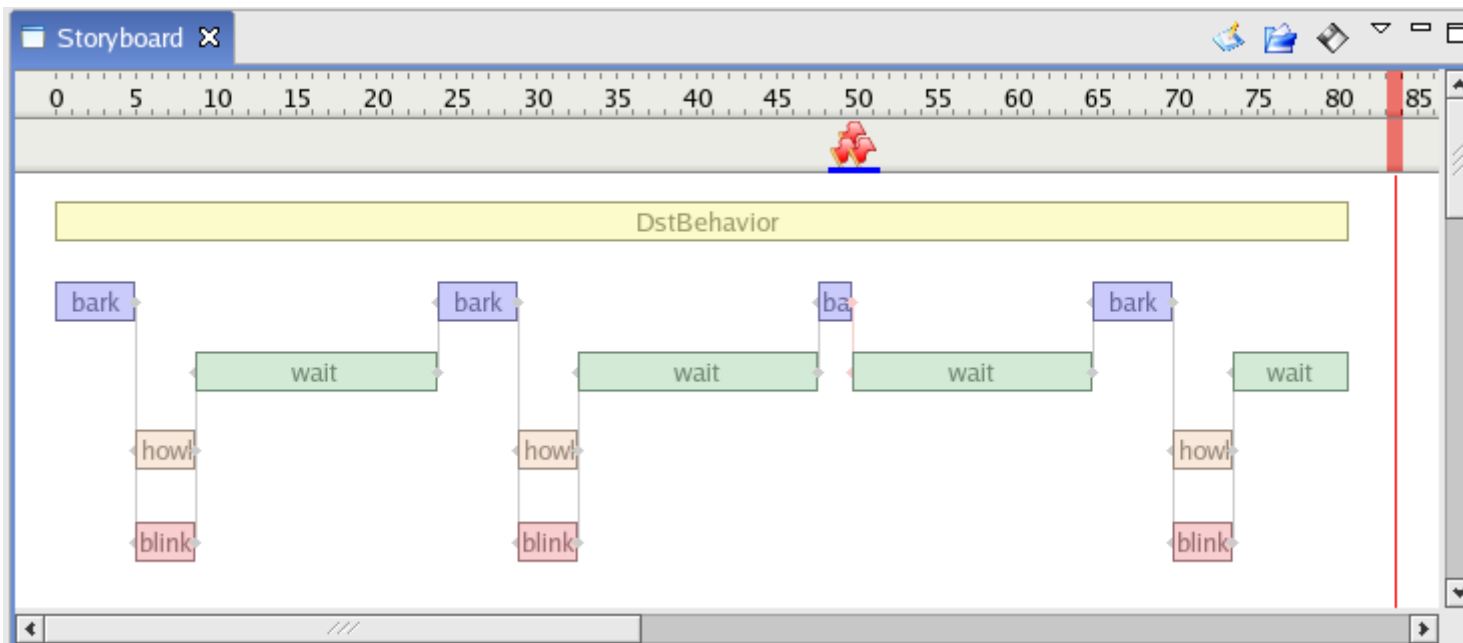
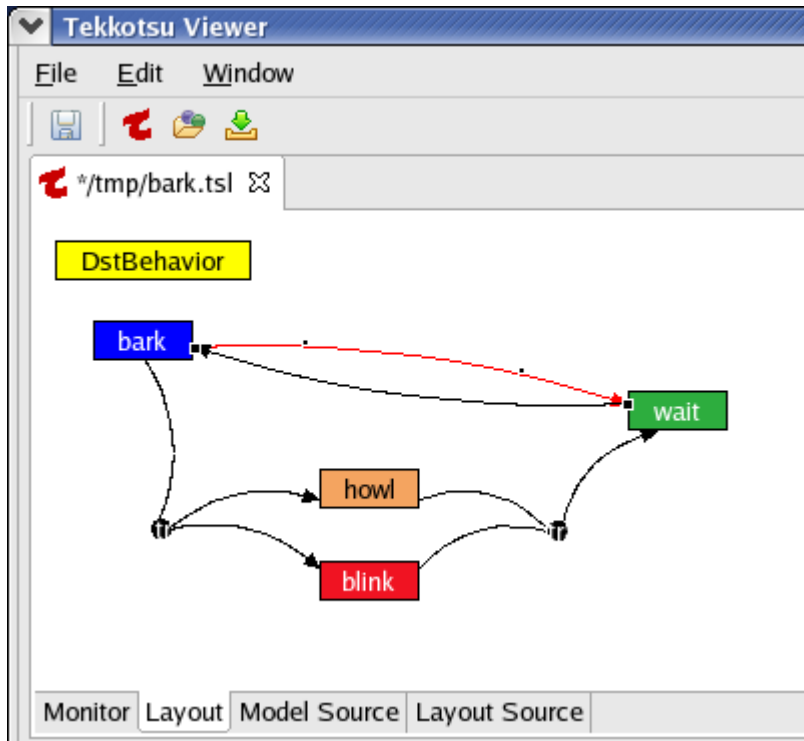
REGISTER_BEHAVIOR(MyMachine);
```

Compiling Your FSM

- The Makefile looks for files with names of form *.fsm and automatically runs them through the state machine compiler, called “stateparser”.
- BarkHowIBlinkBehavior.cc.fsm generates a pure C++ file called BarkHowIBlinkBehavior-fsm.cc.
- The .cc file is stored in:
 ~/project/build/PLATFORM_LOCAL/TARGET_XXX/
- You can run the stateparser directly:

```
stateparser BarkHowIBlinkBehavior.cc.fsm -
```


Storyboard Tool: State Machine Layout



Storyboard Tool: Storyboard Display

The screenshot displays the Tekkotsu Viewer interface, which is used for visualizing and debugging state machines. The main window is titled "Tekkotsu Viewer" and contains a menu bar (File, Edit, Window) and a toolbar with icons for saving, undo, redo, and opening files. The central area shows a state machine model with nodes such as Pink, Follow, Sit, Funny, Timer, Sound, Up, Down, Sniff, Look, and Punch, connected by arrows representing transitions. To the right of the model is a "Properties" panel with a "Runtime View" tab, showing details for the current selection: "Timer" (activate at: 8.885s, deactivate at: 27.0s, type: state), "Timer--:Sit" (fire at: 27.001s, type: transition), and "Sit" (activate at: 27.002s, deactivate at: 27.5s, type: state). Below the model is a "Storyboard" panel with a timeline from 0 to 60 seconds. A red vertical line indicates the current time, which is approximately 27.5 seconds. The storyboard shows the sequence of states and transitions over time, with a green bar representing the "Timer" state and a blue bar representing the "Sit" state. To the right of the storyboard is an "Image Preview" panel, which is currently empty.

Storyboard Tool: Snapshots

The screenshot displays the Tekkotsu Viewer application interface, which is used for developing and running state machines. The interface is divided into several panels:

- Top Panel:** Contains the menu bar (File, Edit, Window) and a toolbar with icons for file operations. Below this are three browser tabs showing file paths: `*/afs/cs.cmu.edu/user/dst/S...`.
- Host Configuration:** A section on the right with fields for "Host" (localhost) and "Port" (10080). Below these are a "Name" field (Explore State Machine) and buttons for "Download Model", "New Trace", and a refresh icon.
- Storyboard View (Bottom):** A timeline-based view showing the execution of a state machine. A yellow bar at the top is labeled "Logging Test". Below it, a sequence of states is shown: "Waiting" (green), "Image" (blue), "Webcam" (red), "Message" (purple), "Message" (purple), and "Image" (blue). Each state is connected to a "Waiting" state below it. A red vertical line indicates the current time on the timeline, which has markers at 0, 5, 8.43, 8.91, 15, 18.2, 20, 22.17, 25.18, 30, 35.99, 35.99, 40, 45.35, and 45.41.
- Properties Panel (Right):** Shows the "Runtime View" for the current selection at time :9.491s. It lists the following state information:
 - Image:Image**
 - record at: 8.457s
 - type: image
 - Waiting**
 - activate at: 8.495000000000000
 - deactivate at: 18.201s
 - type: state
 - Logging Test**
 - activate at: 0.0s
 - deactivate at: 57.206s
- Image Preview (Bottom Right):** A window showing a live video feed from a webcam, displaying a room with a computer monitor and other equipment.