State Machines

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

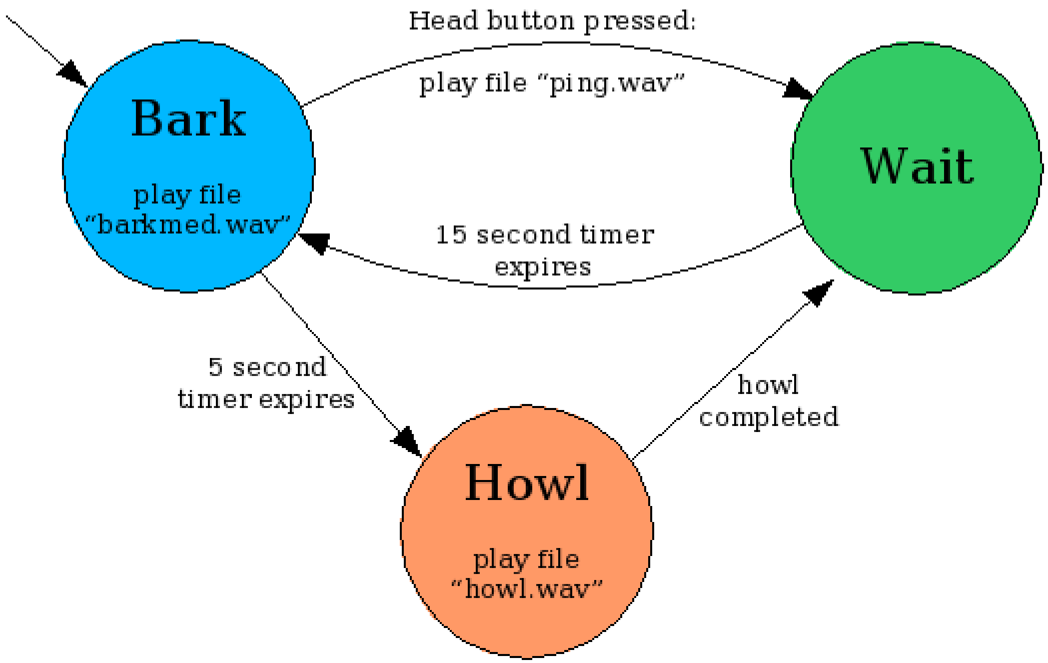
Carnegie Mellon
January 2015

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.

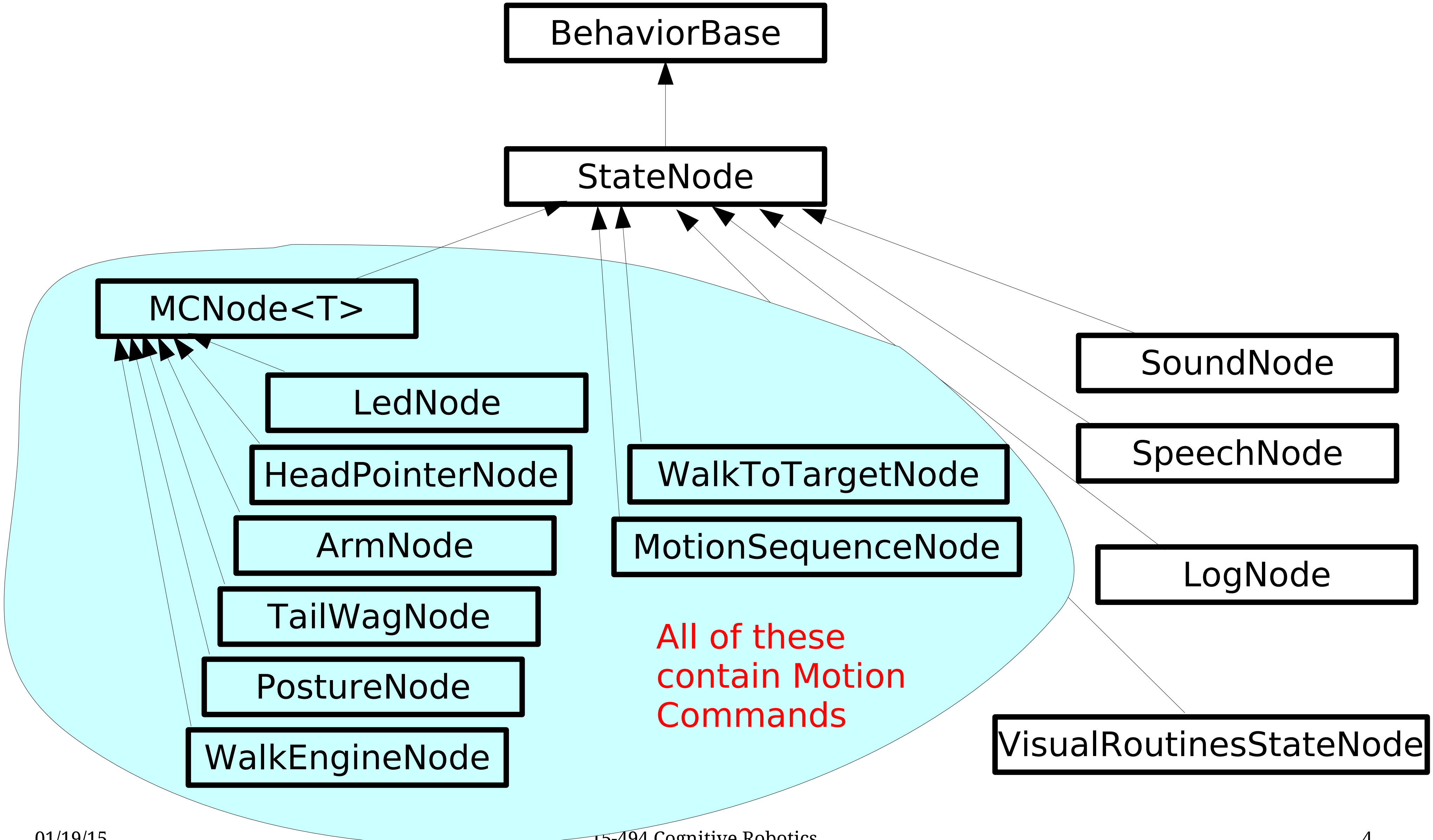
Basic Idea

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

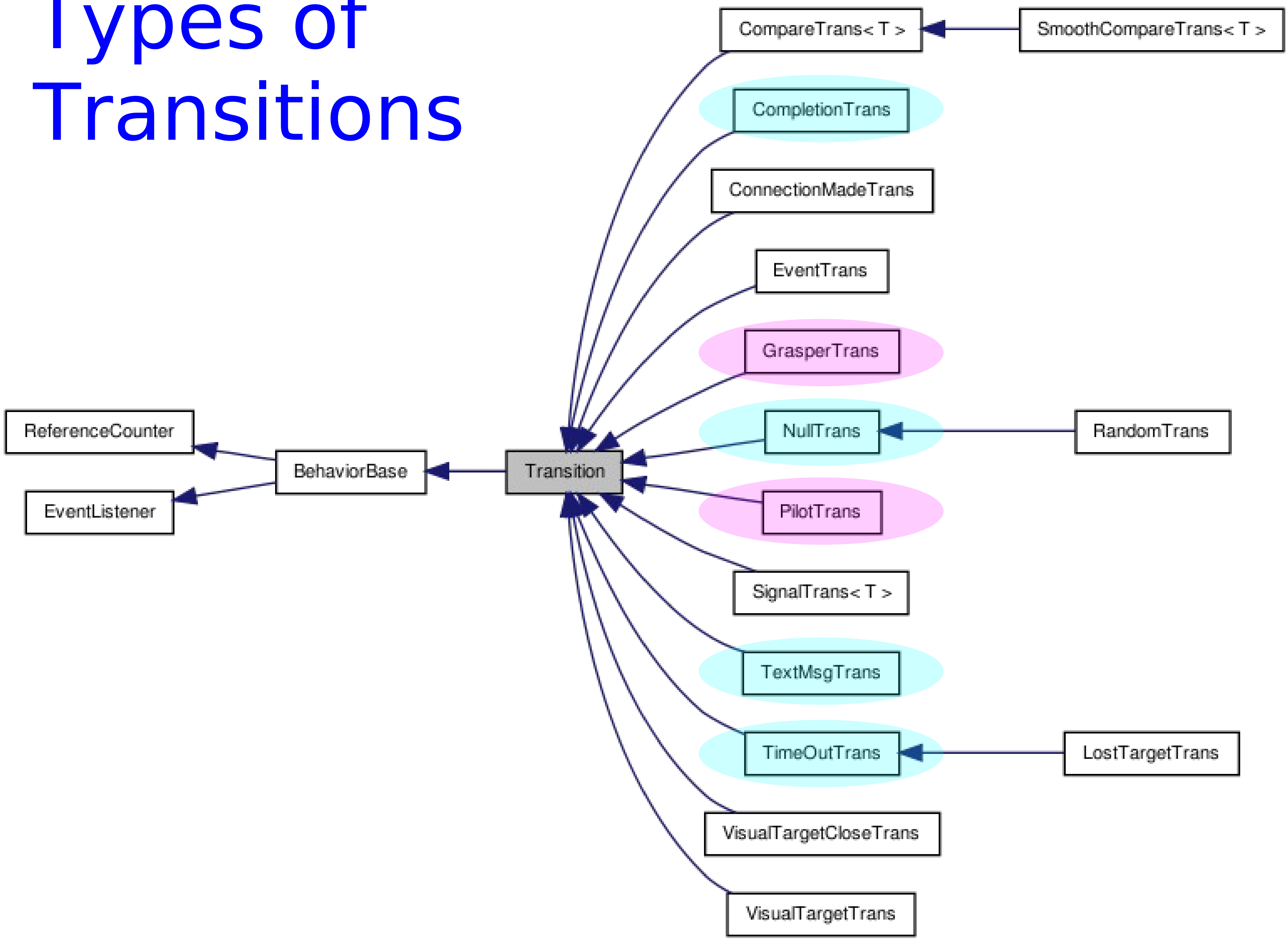


Types of State Nodes

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



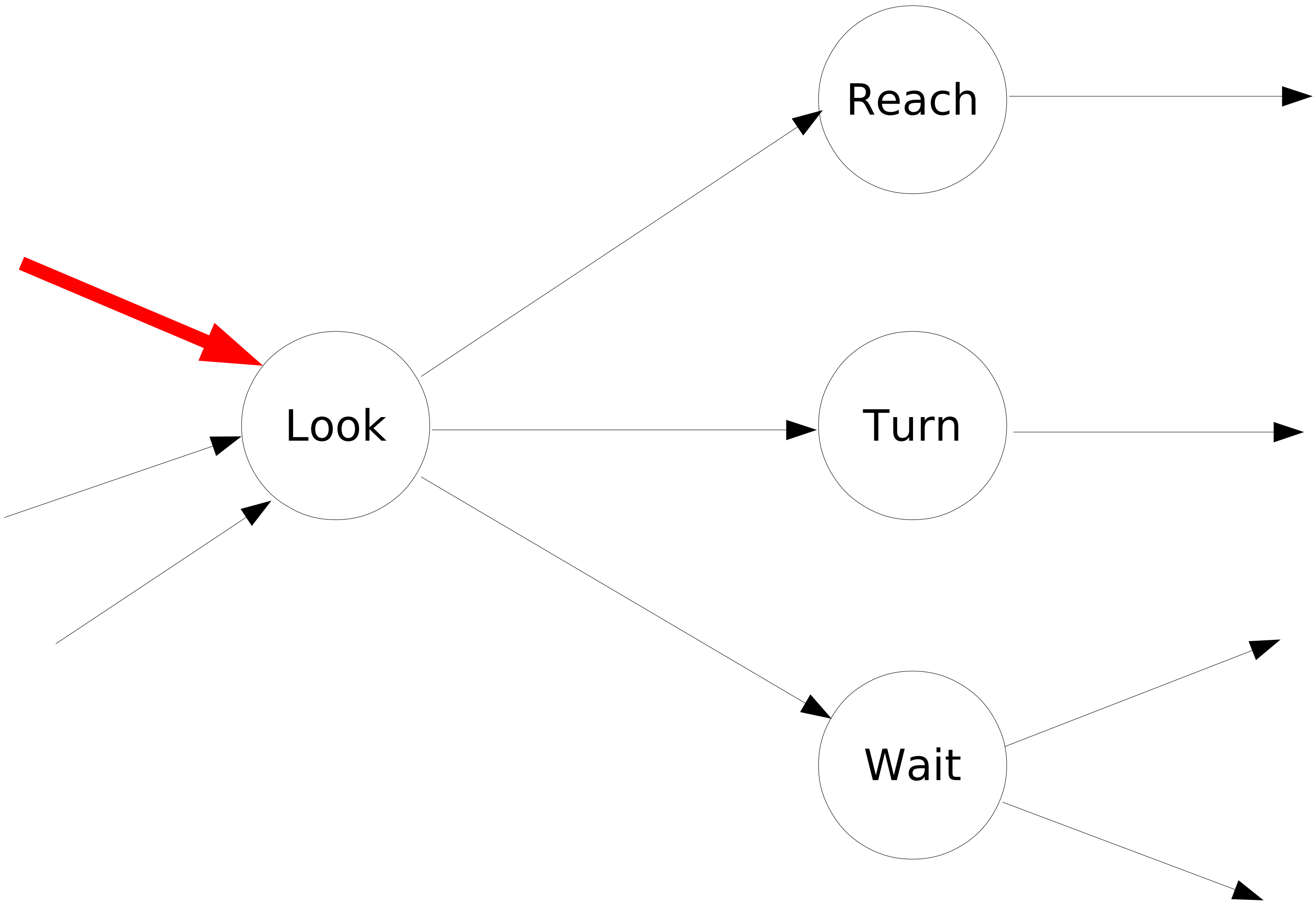
Types of Transitions



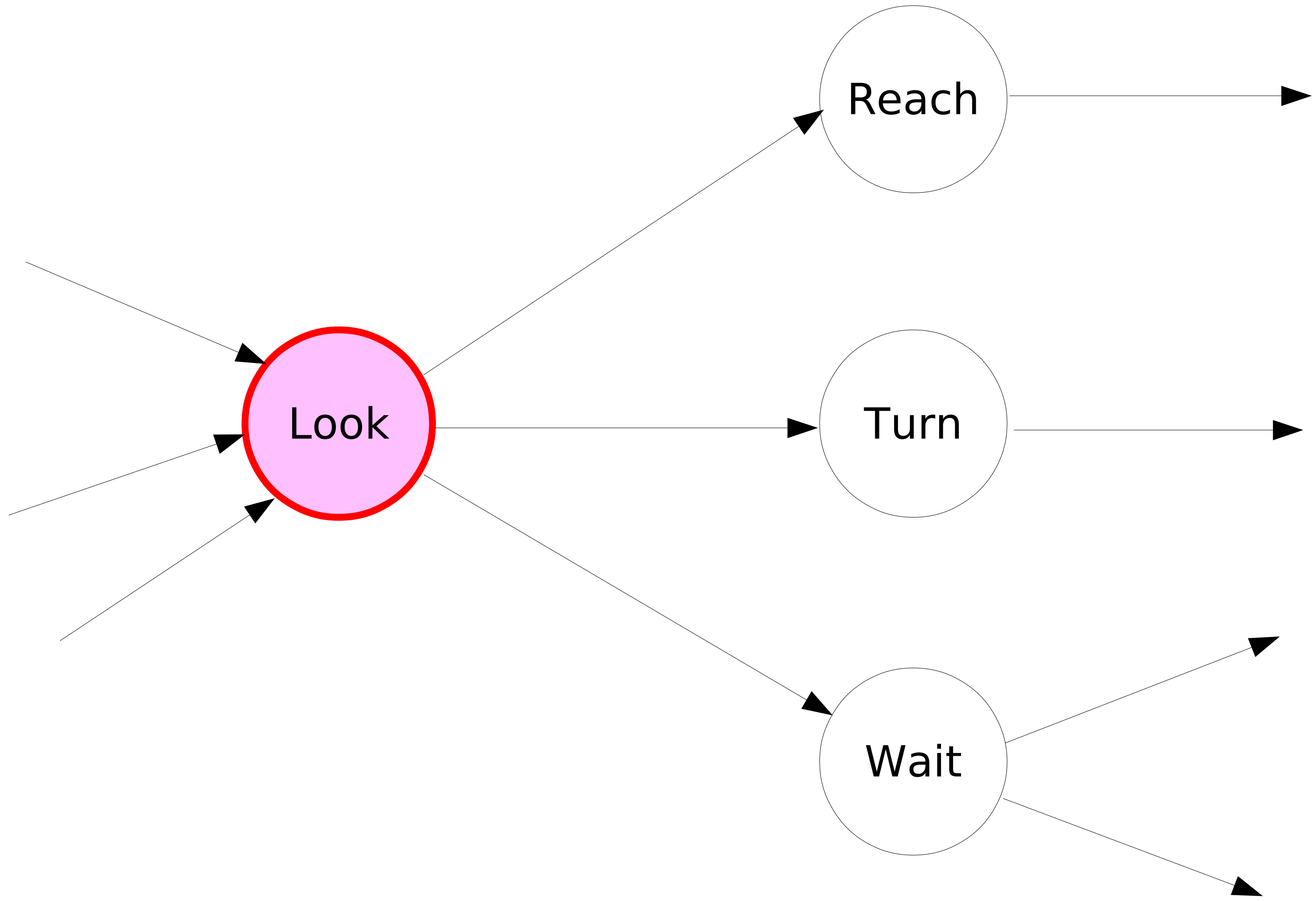
Both State Nodes and Transitions Are Behaviors

- 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).

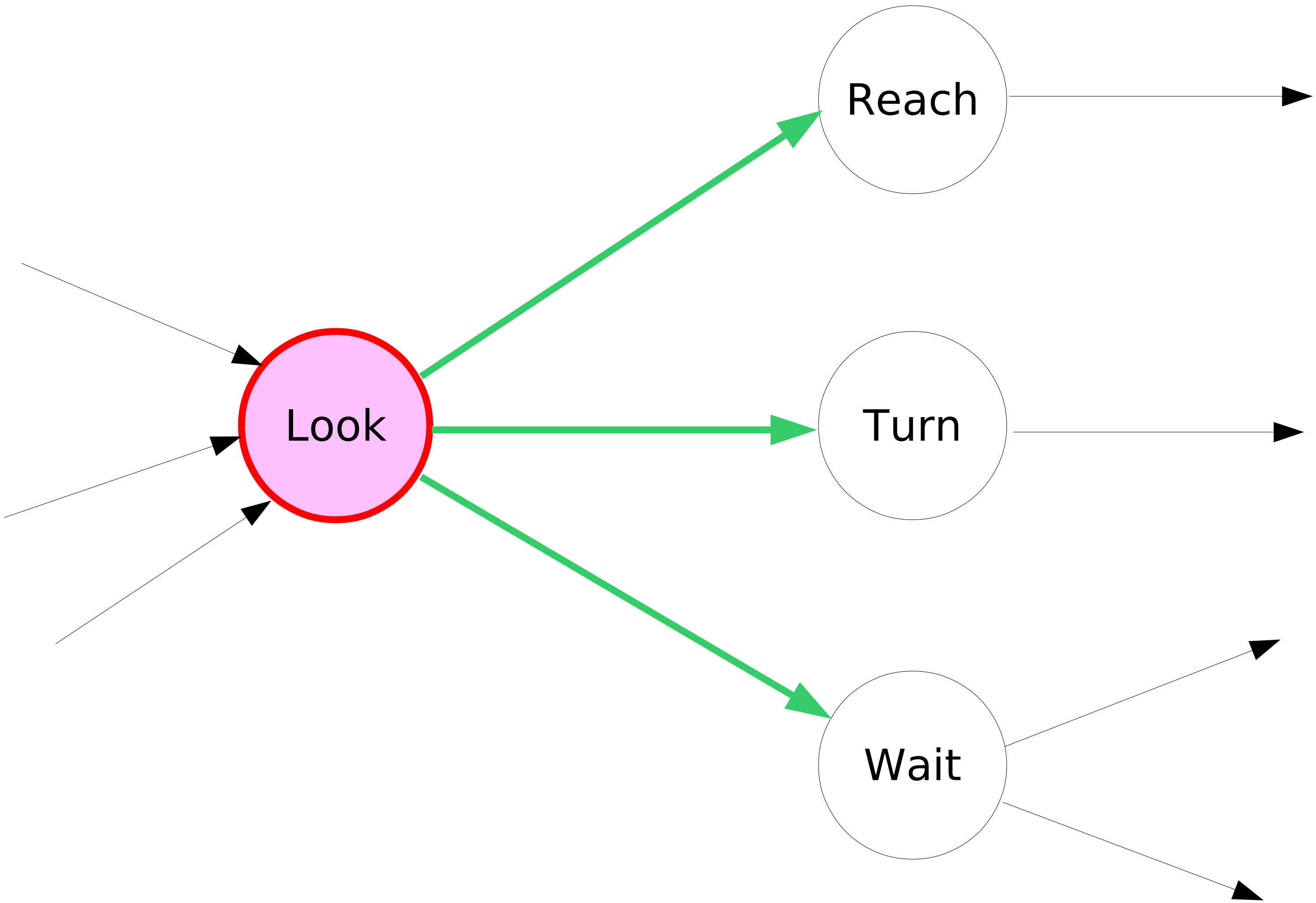
Transition firing activates state node Look.



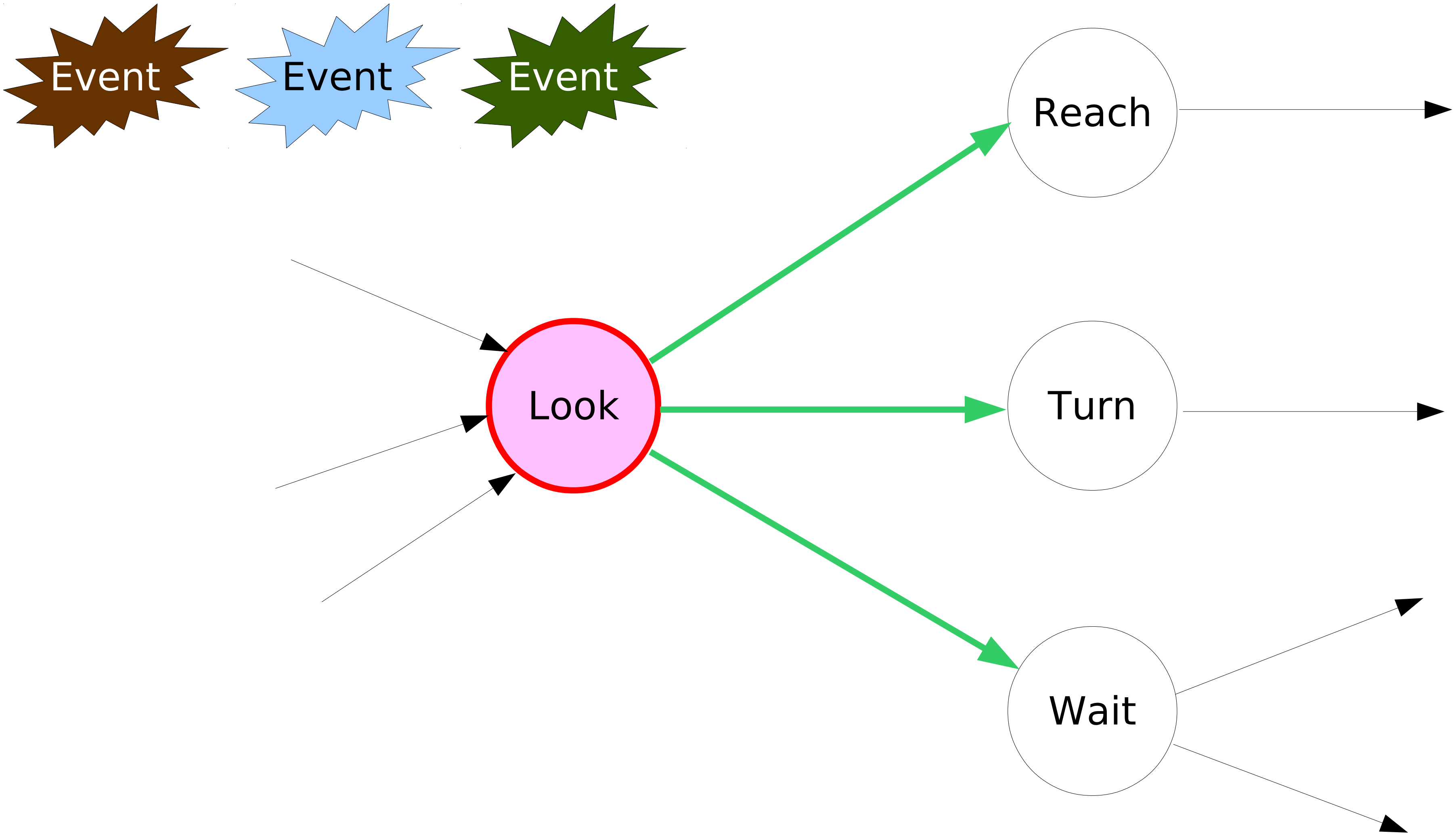
Look's start() calls StateNode::start().



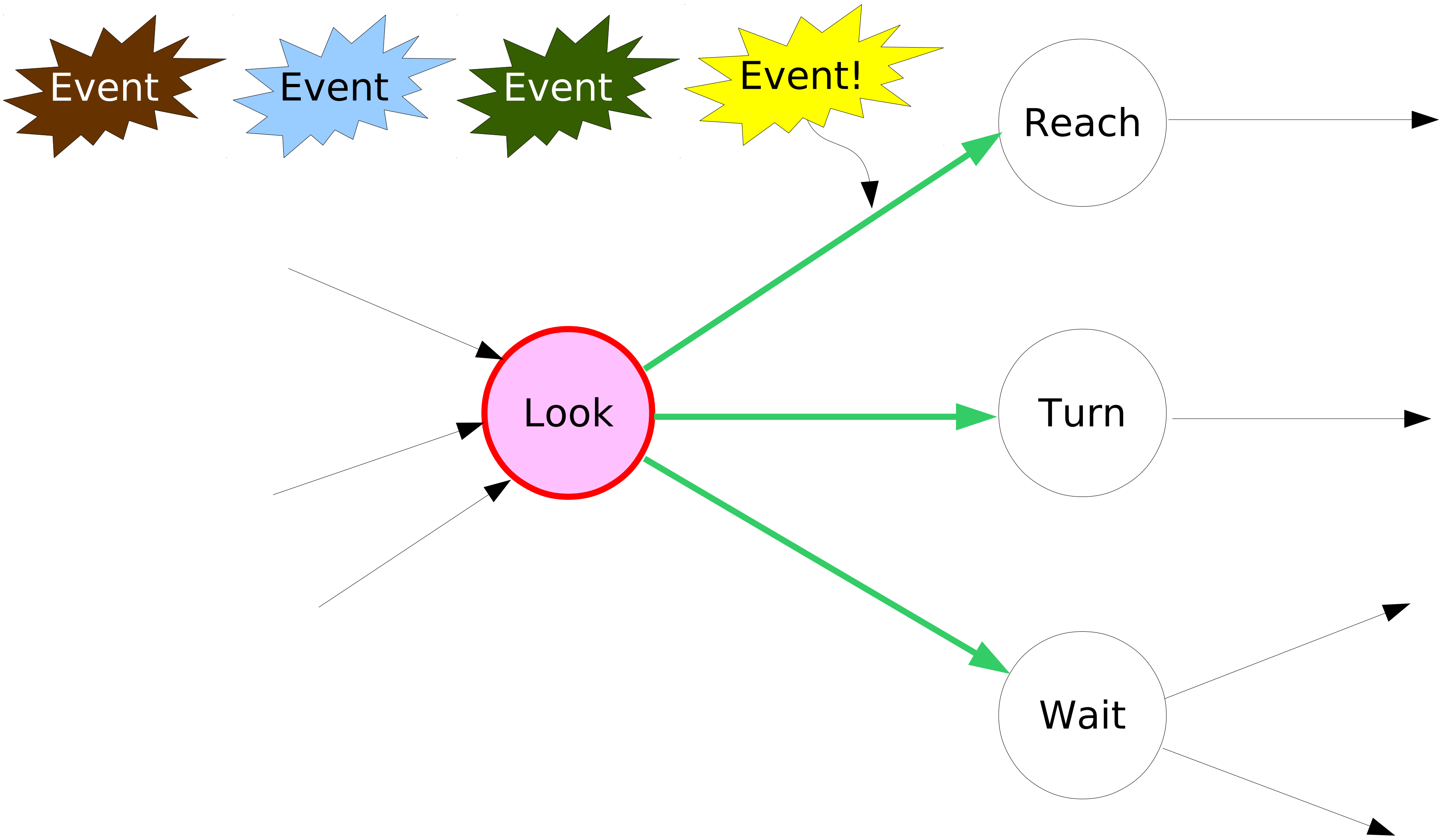
Outgoing transitions become active and begin listening for events.



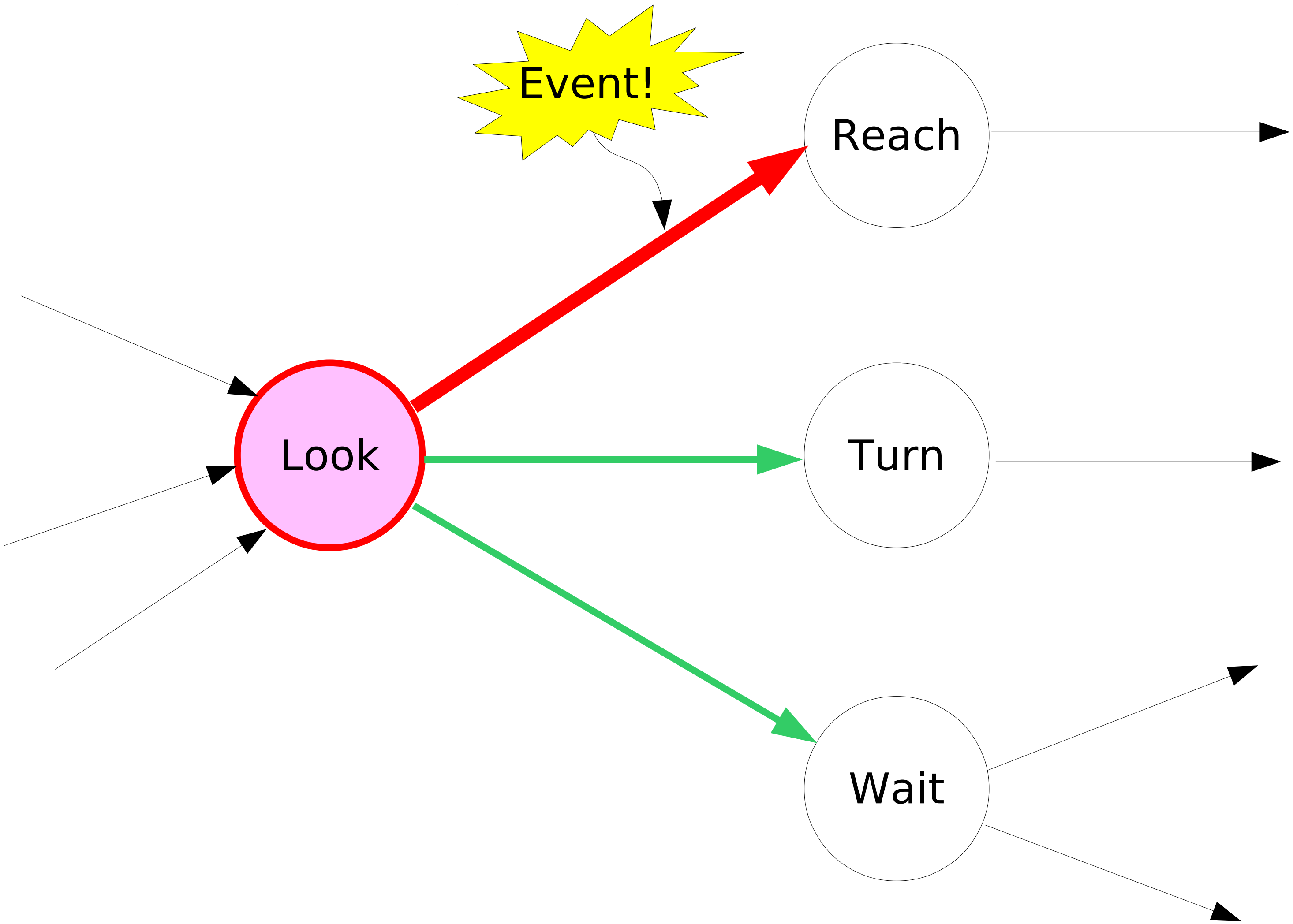
Random things happen....



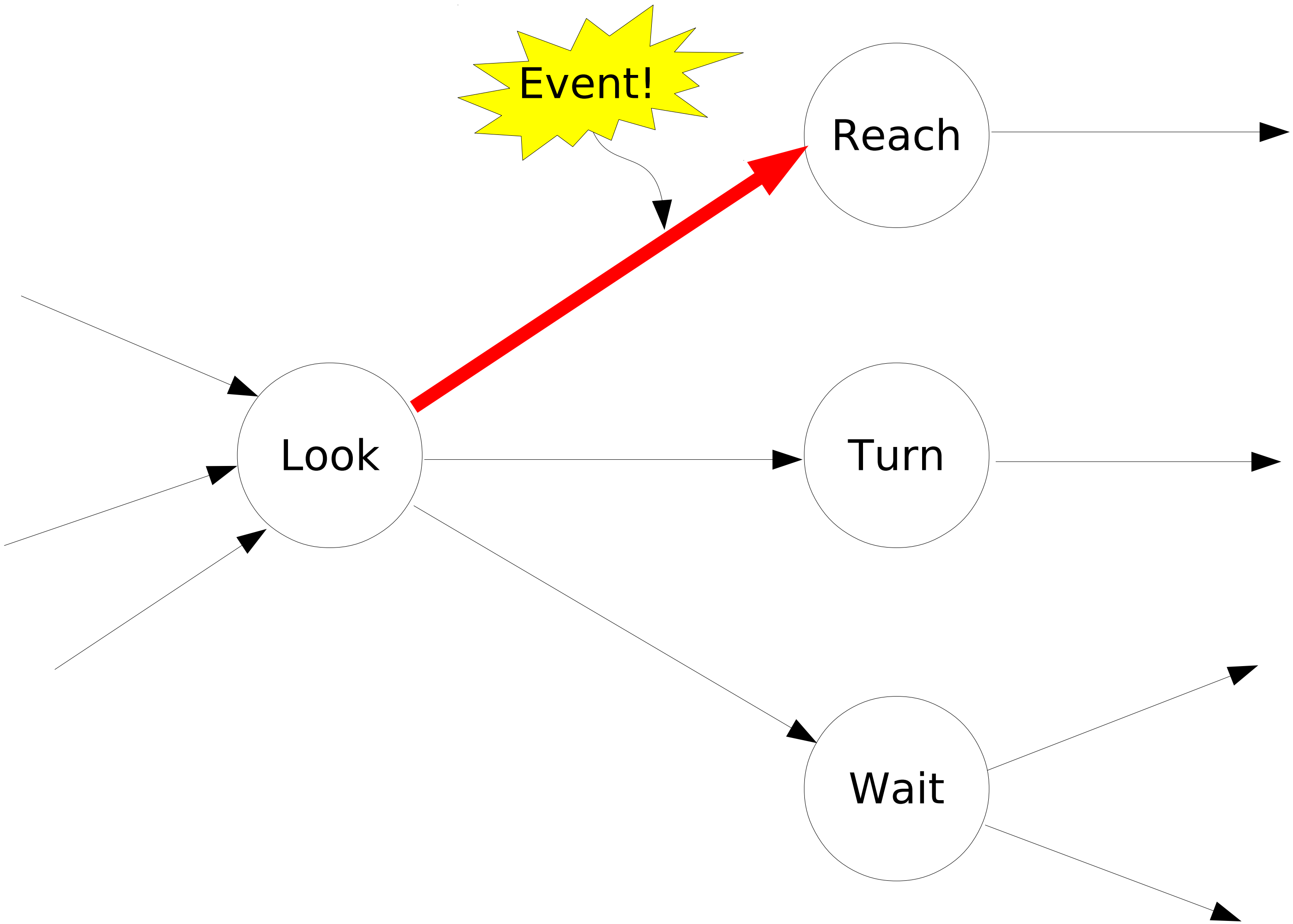
And then, something we've been looking for...



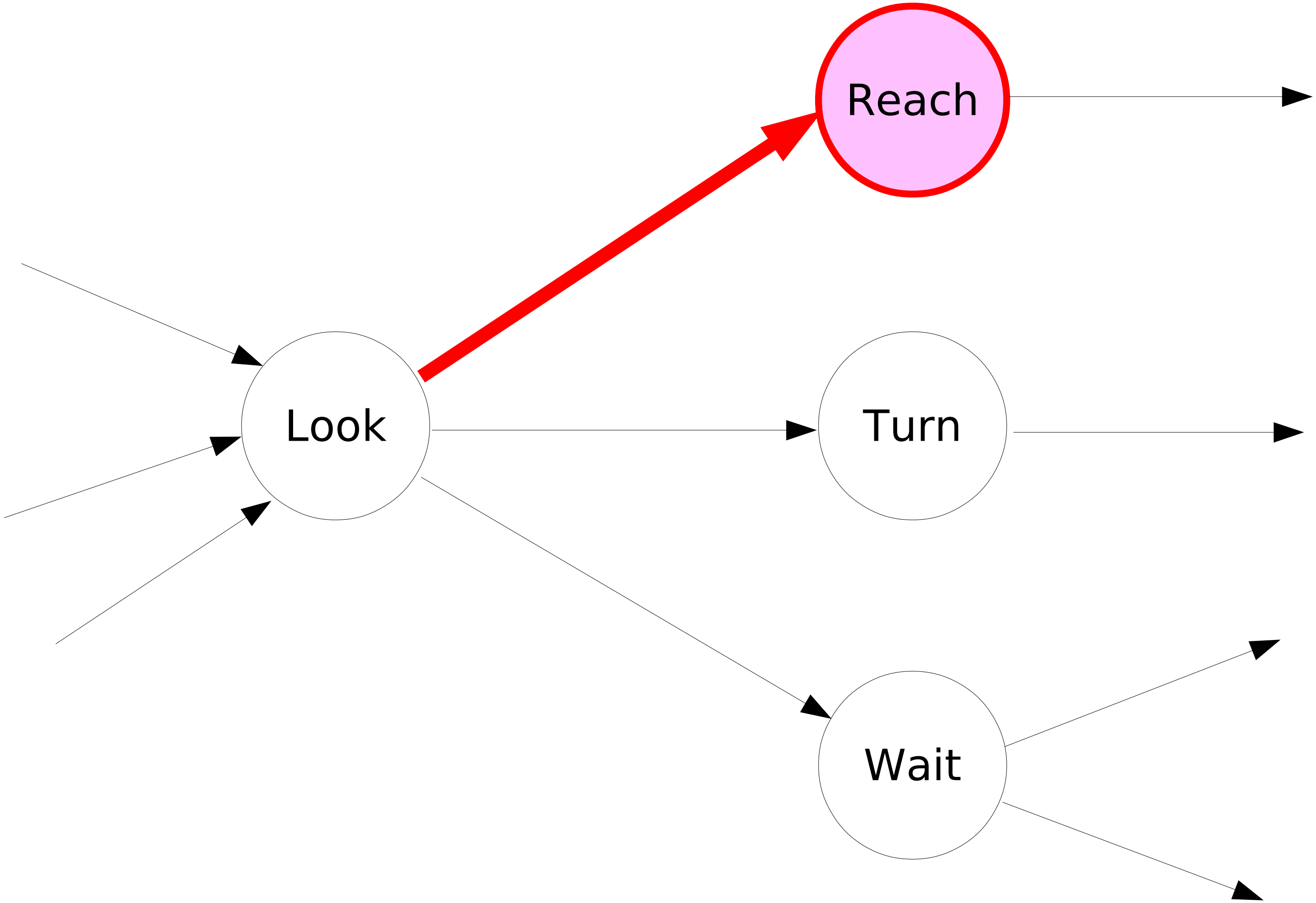
Transition decides to fire.



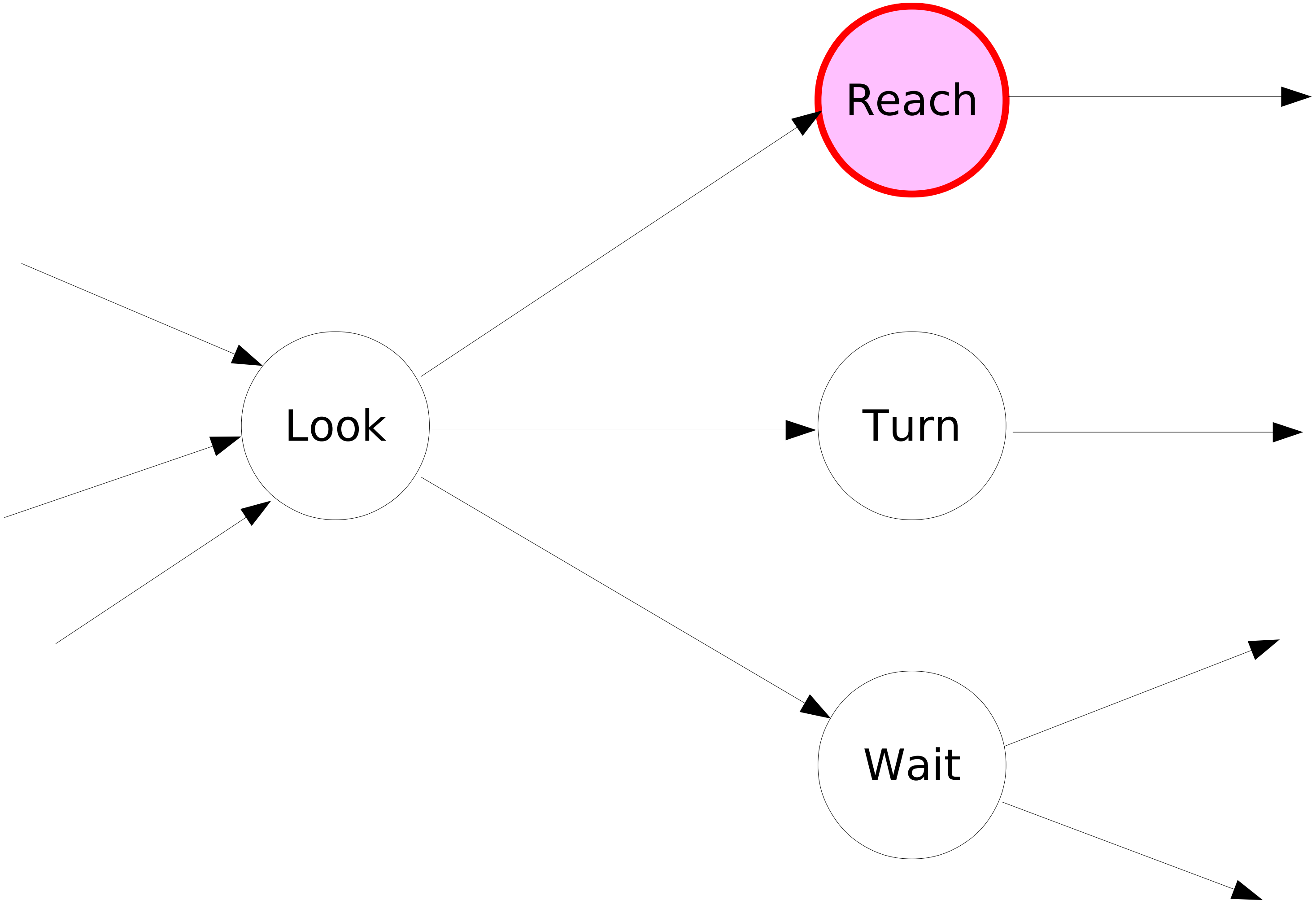
Transition deactivates the source node, Look.



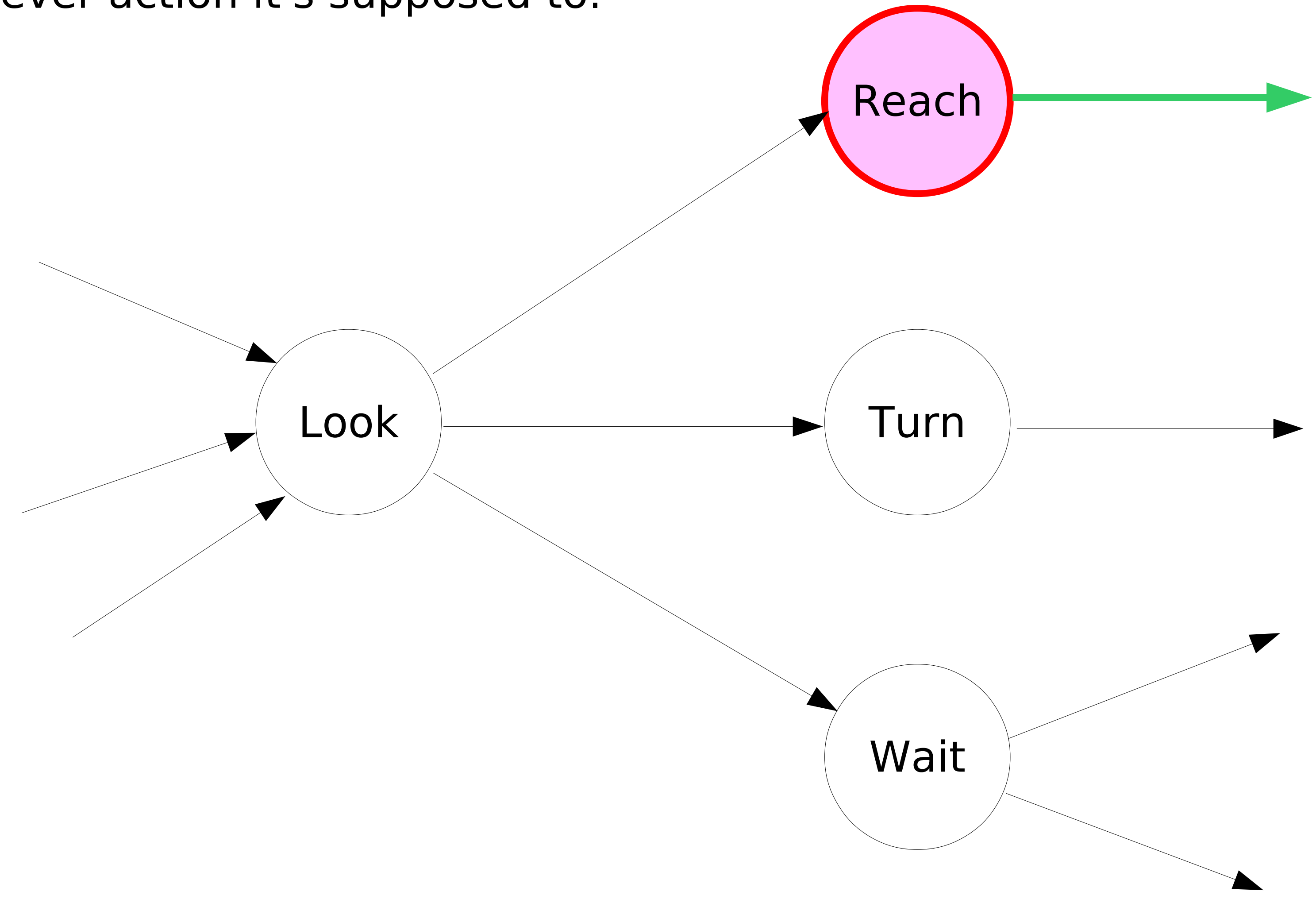
Transition activates the target node, Reach.



Transition deactivates.



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

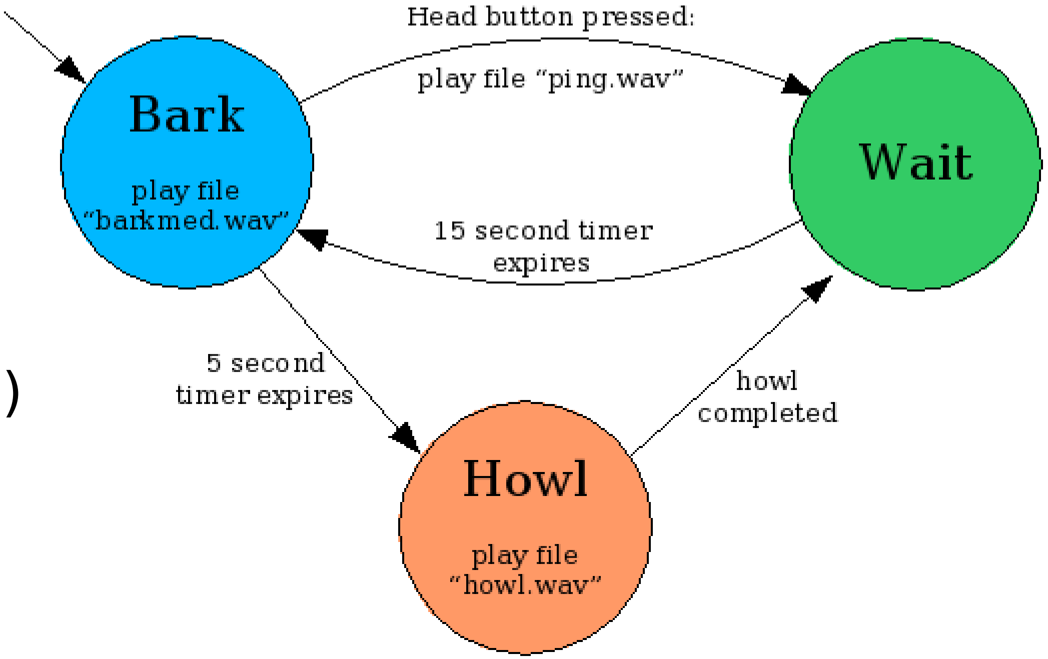


State Machine Compiler

- 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 {

  $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.

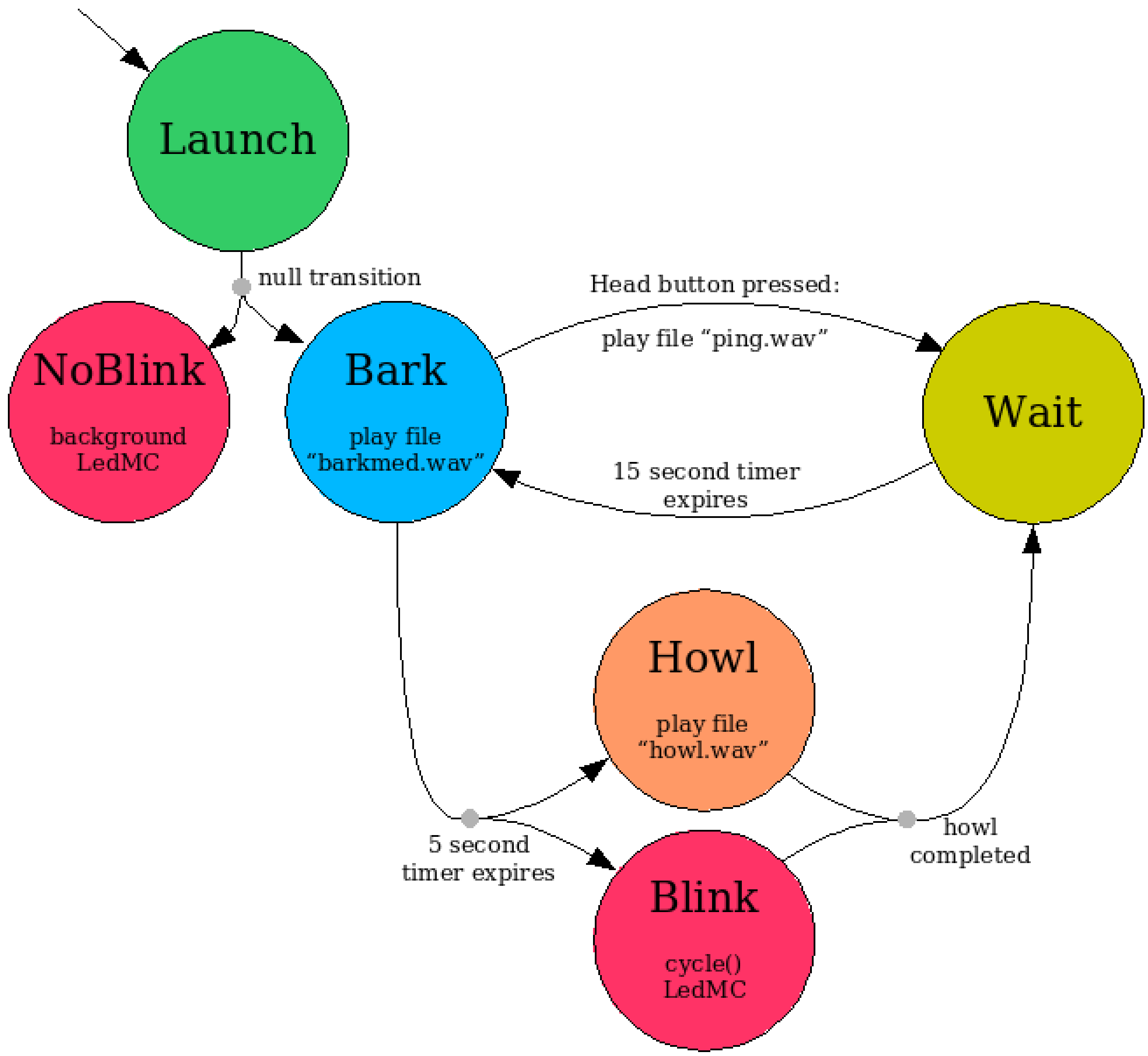
- It's good to chain if a node has only one transition.
- Example:

```
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.

file: BarkHowlBlinkBehavior.cc.fsm

```
$setupmachine{
  // Annoying dog with blinking LEDs

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

  noblink:  deferred


  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.

file: BarkHowlBlinkBehavior.cc.fsm

```
$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

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

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

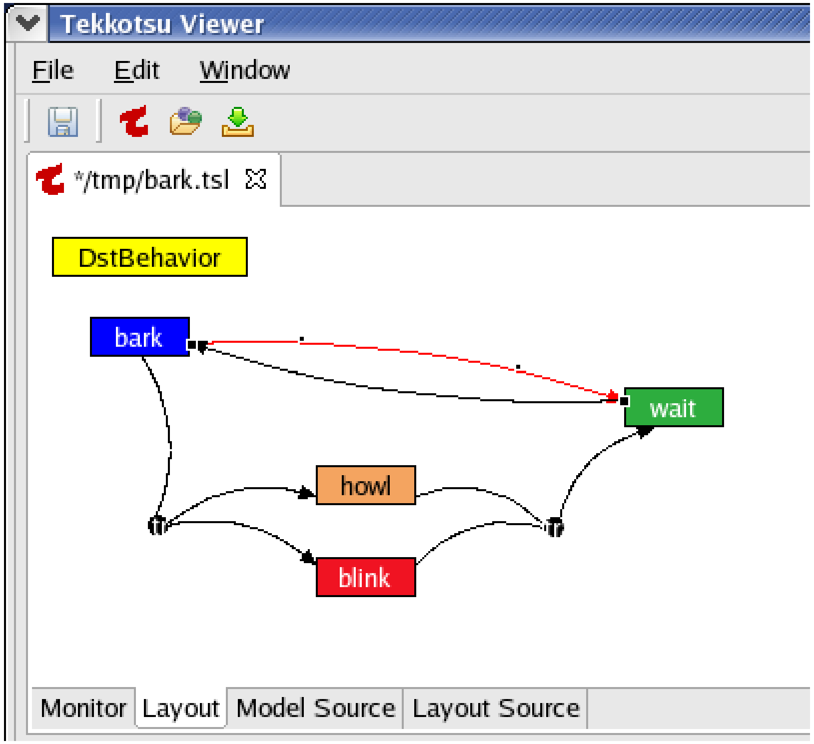
```
#include "Behaviors/StateMachine.h"

$nodeclass MyMachine {

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

  $nodeclass Sendoff : SpeechNode : doStart {
    textstream << "So long!" << endl;
  }

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

}

REGISTER_BEHAVIOR(MyMachine);
```

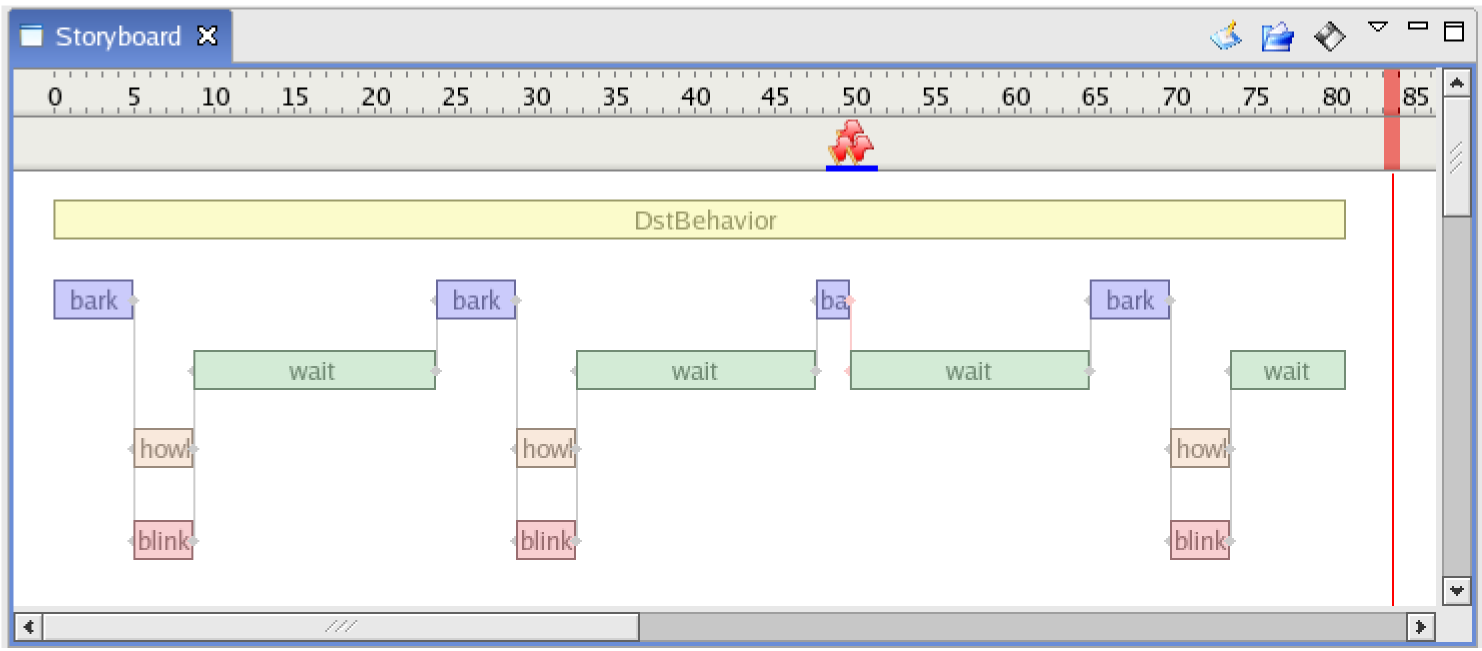
Compiling Your FSM

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

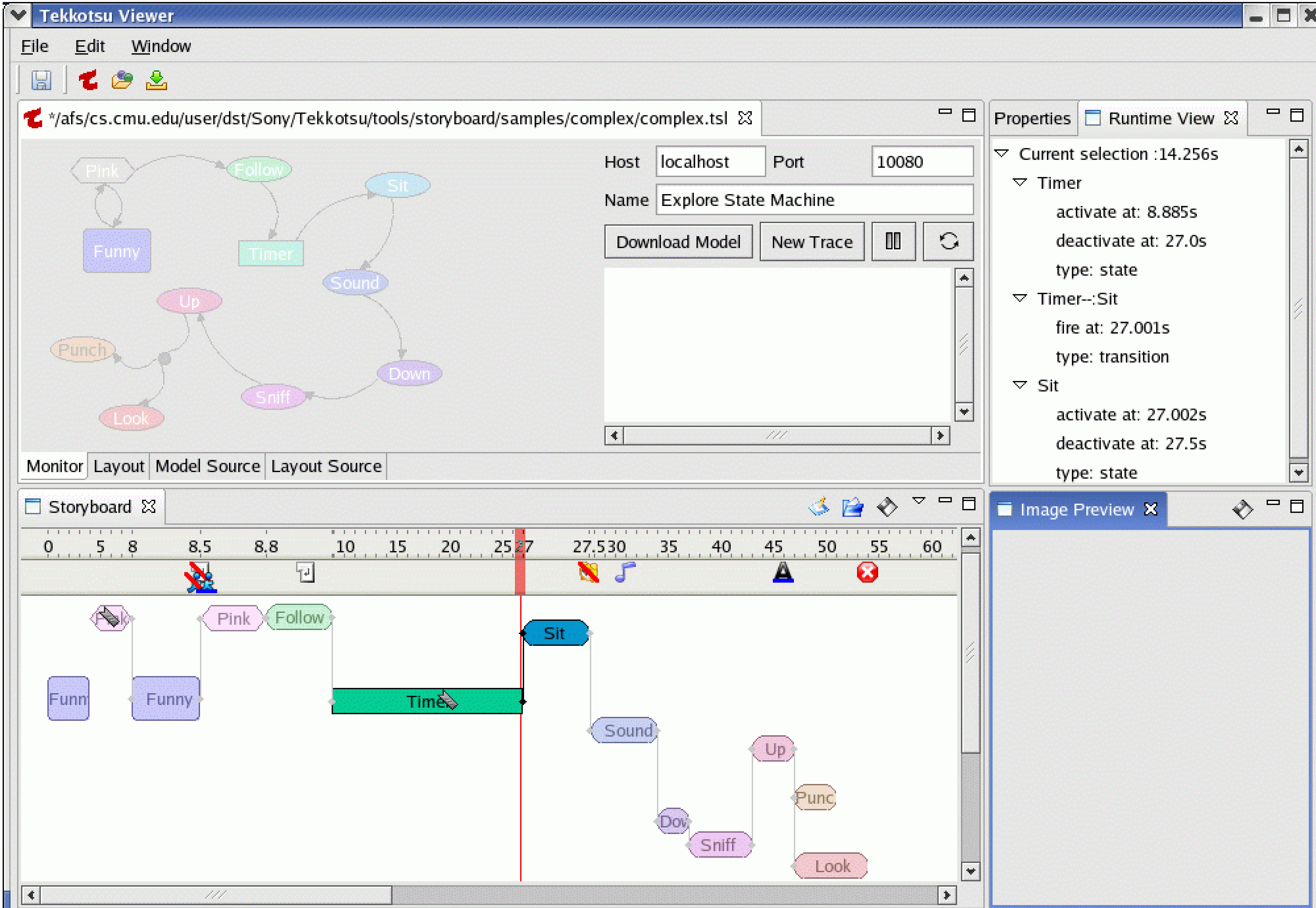
  stateparser BarkHowlBlinkBehavior.cc.fsm –

Storyboard Tool: State Machine Layout





Storyboard Tool: Storyboard Display



Storyboard Tool: Snapshots