

State Machines

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

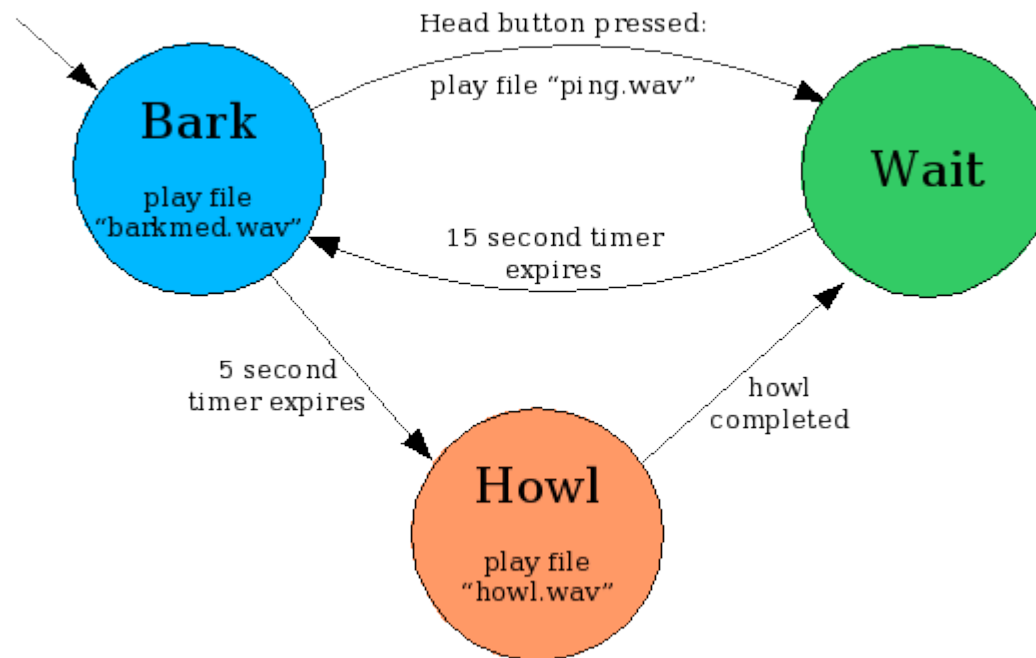
Carnegie Mellon
January 2011

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.

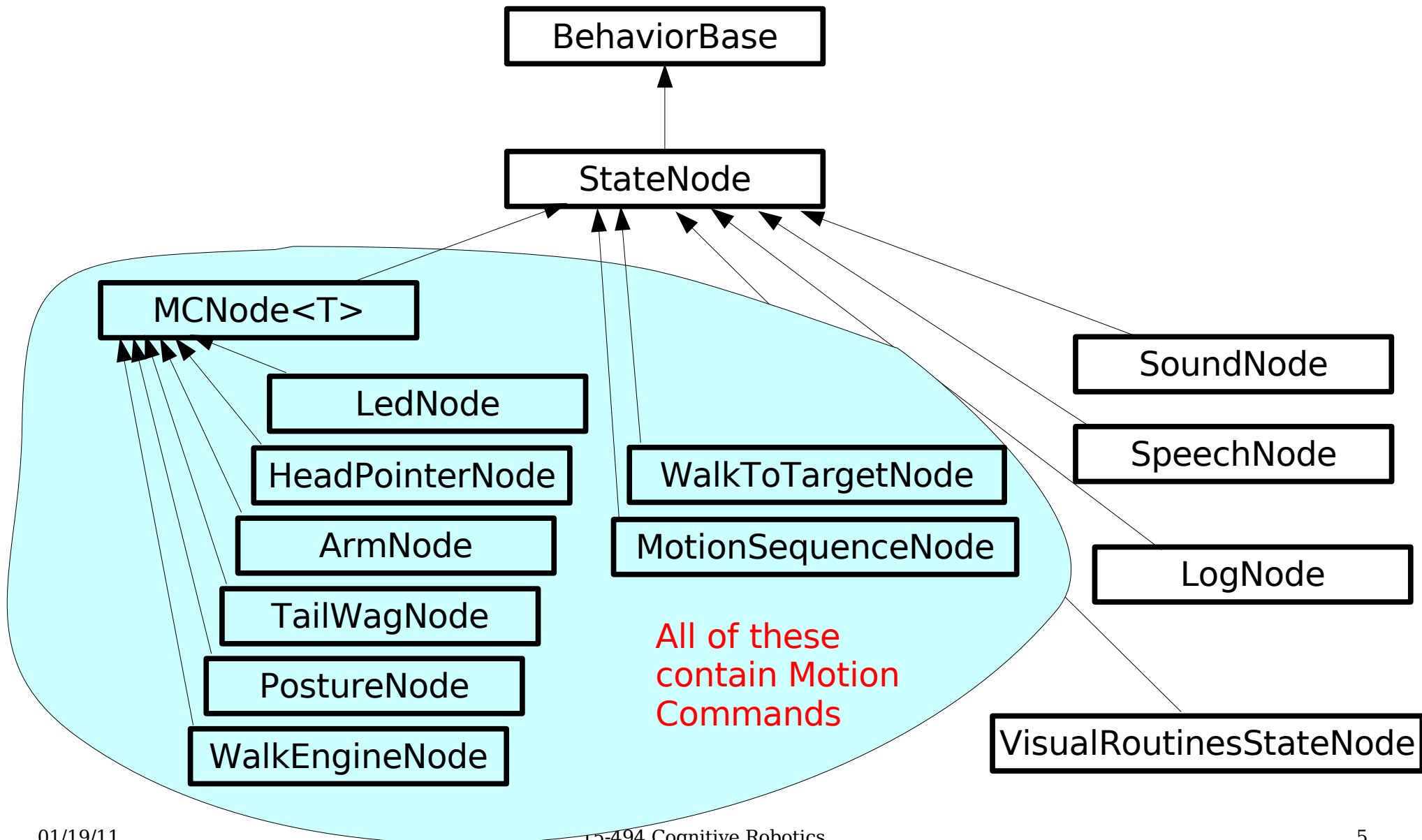


Tekkotsu State Nodes

- In Tekkotsu, state machine nodes are *behaviors*.
- StateNode is a child of BehaviorBase.
- To enter a state, call its start() method, which will call its doStart() method if one has been supplied.
- To leave a state, call its stop() method.
- StateNodes can listen for and process events just like any other behavior.

Types of State Nodes

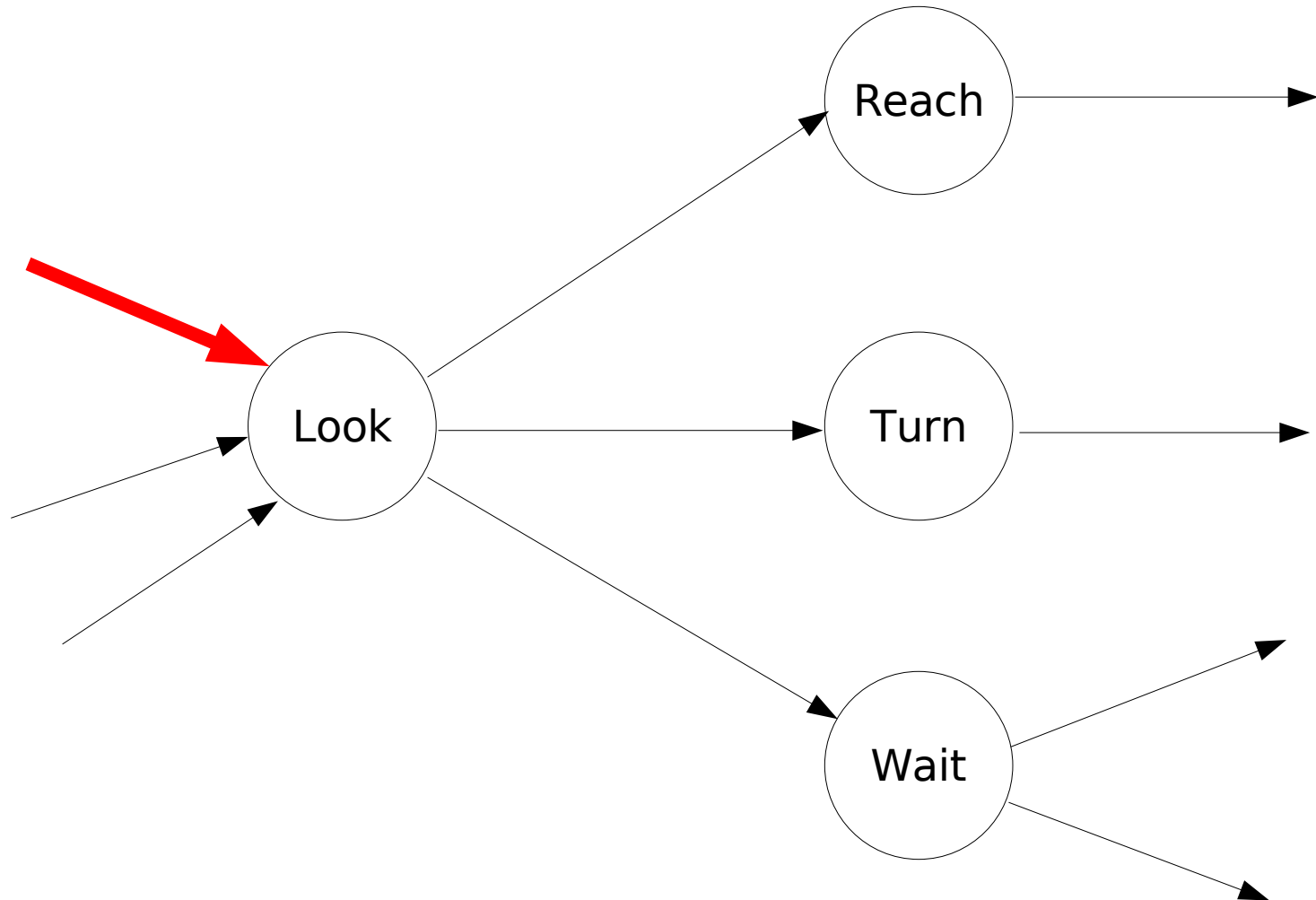
- State nodes encapsulate complex actions, such as creating and launching a motion command.



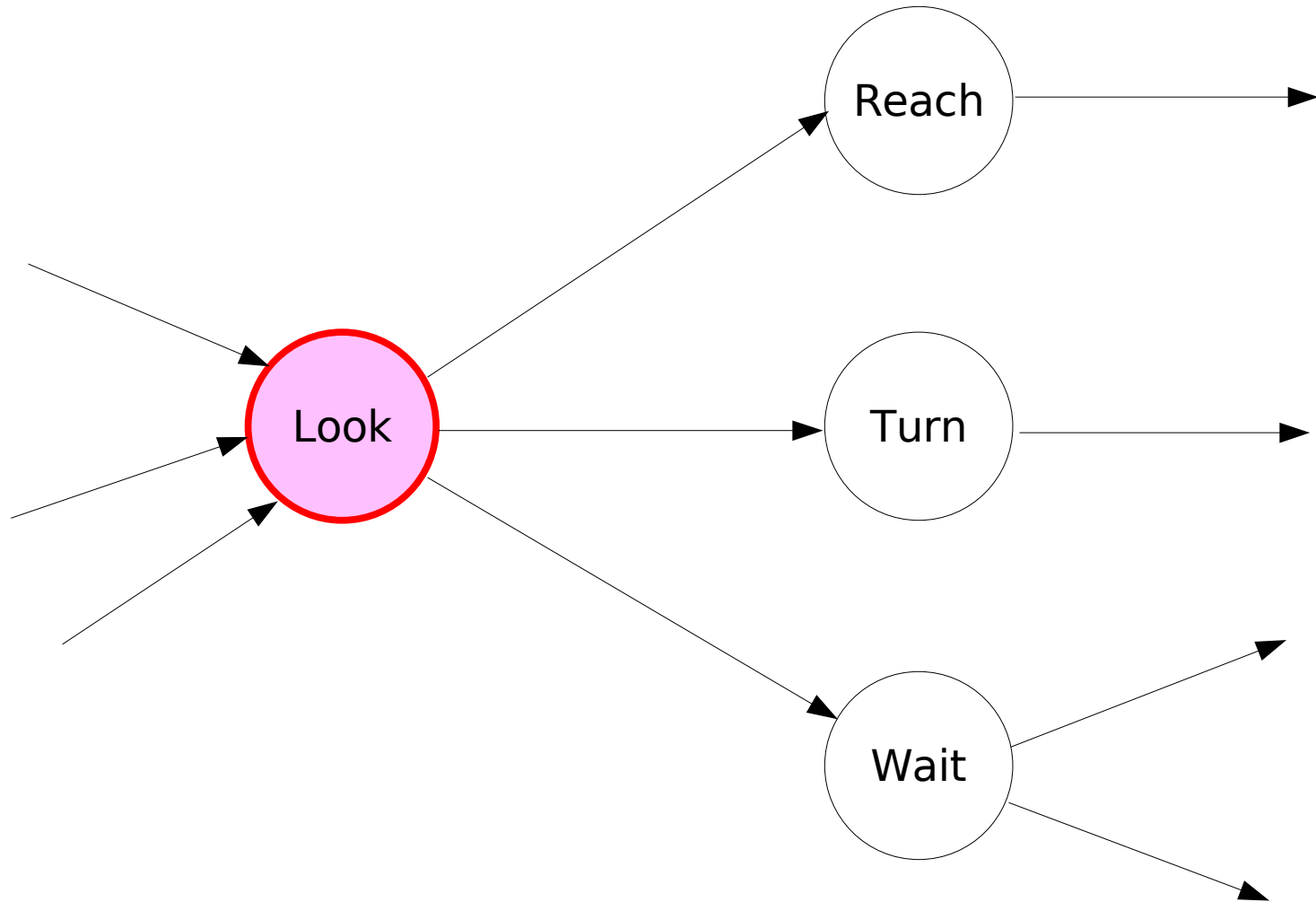
Transitions

- Transitions in Tekkotsu are also behaviors.
 - Transition and StateNode are *both* subclasses of BehaviorBase.
- A transition's start() 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 destination node(s).

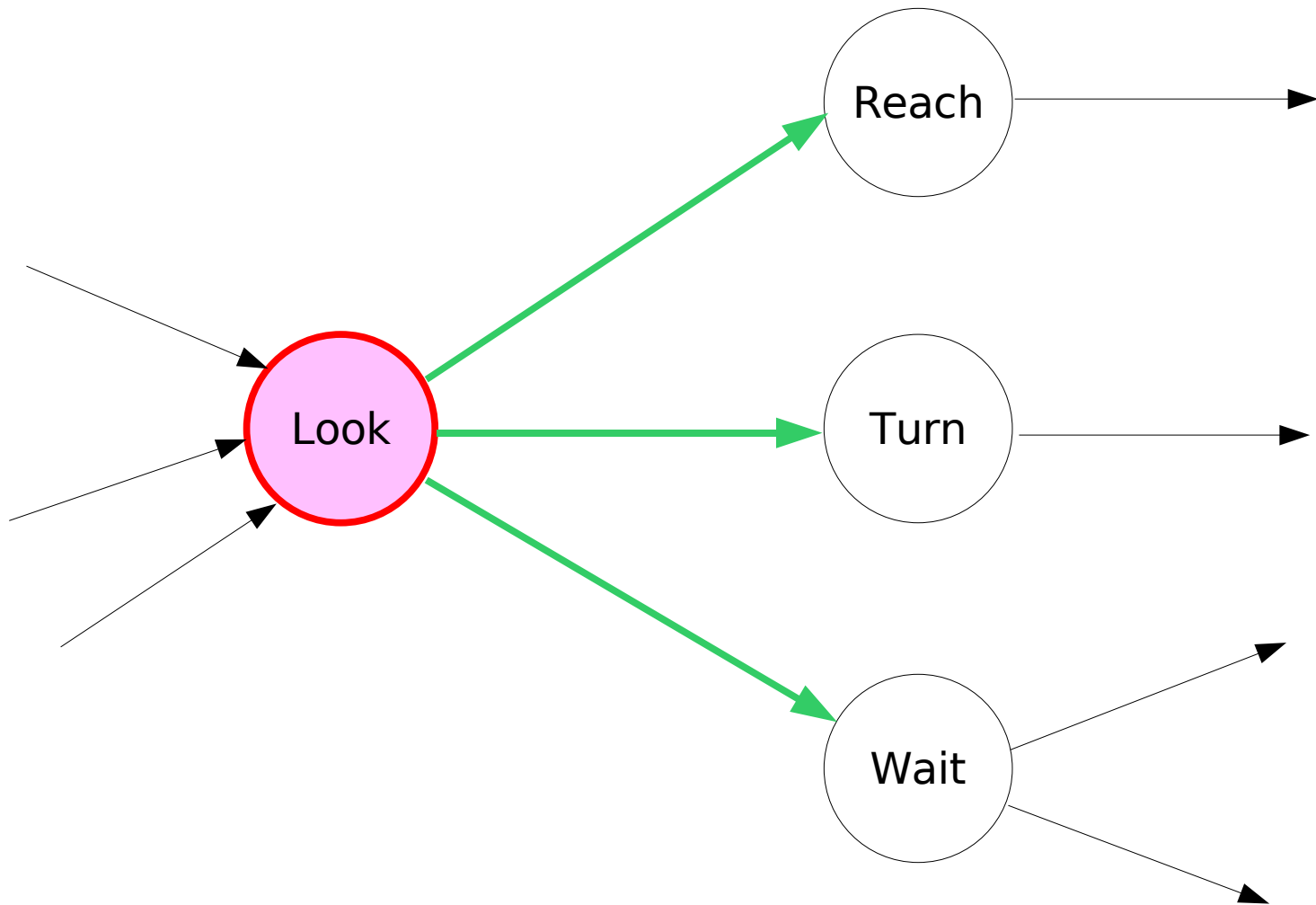
Transition firing activates state node Look.



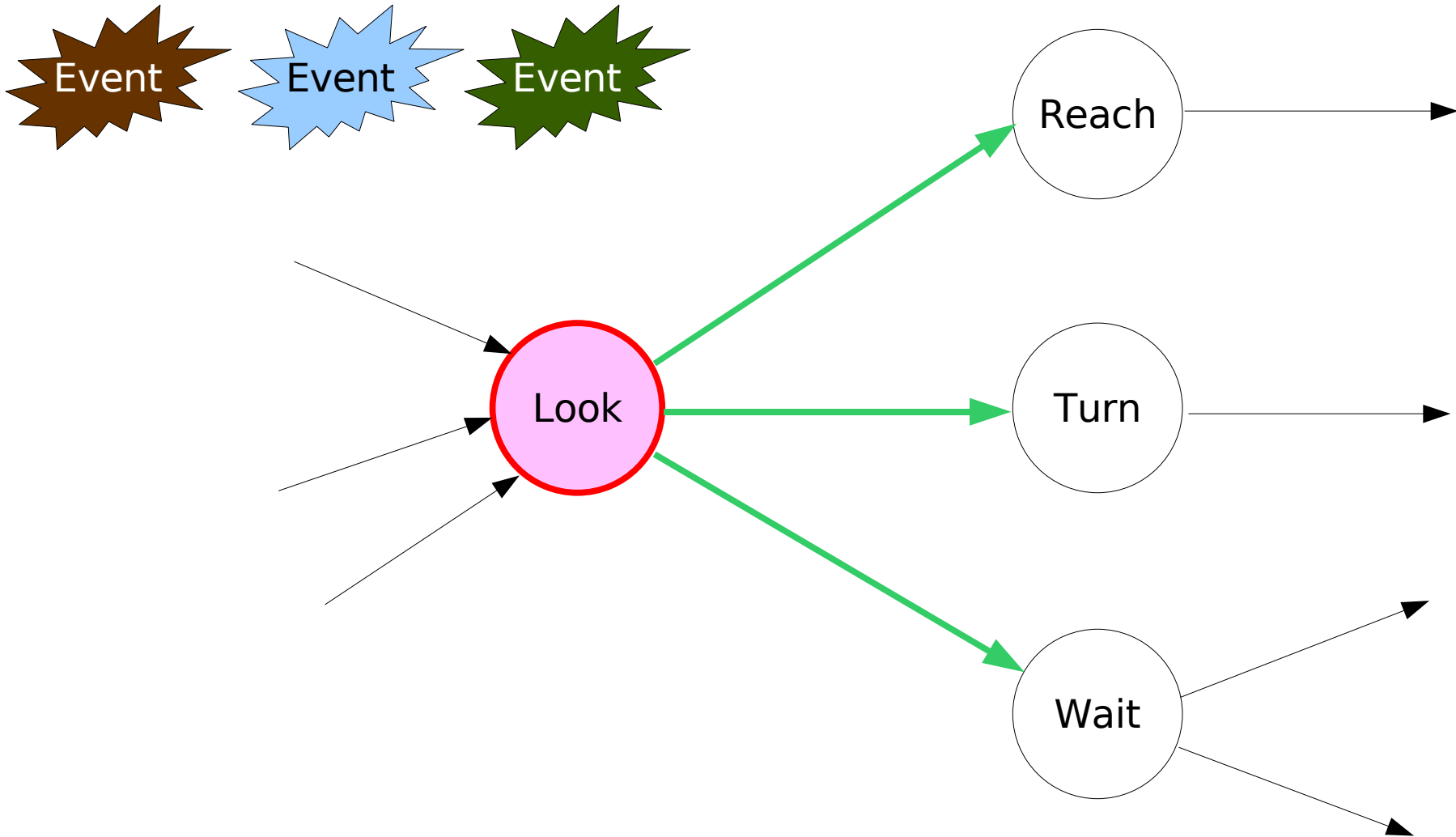
Look's start() calls StateNode::start().



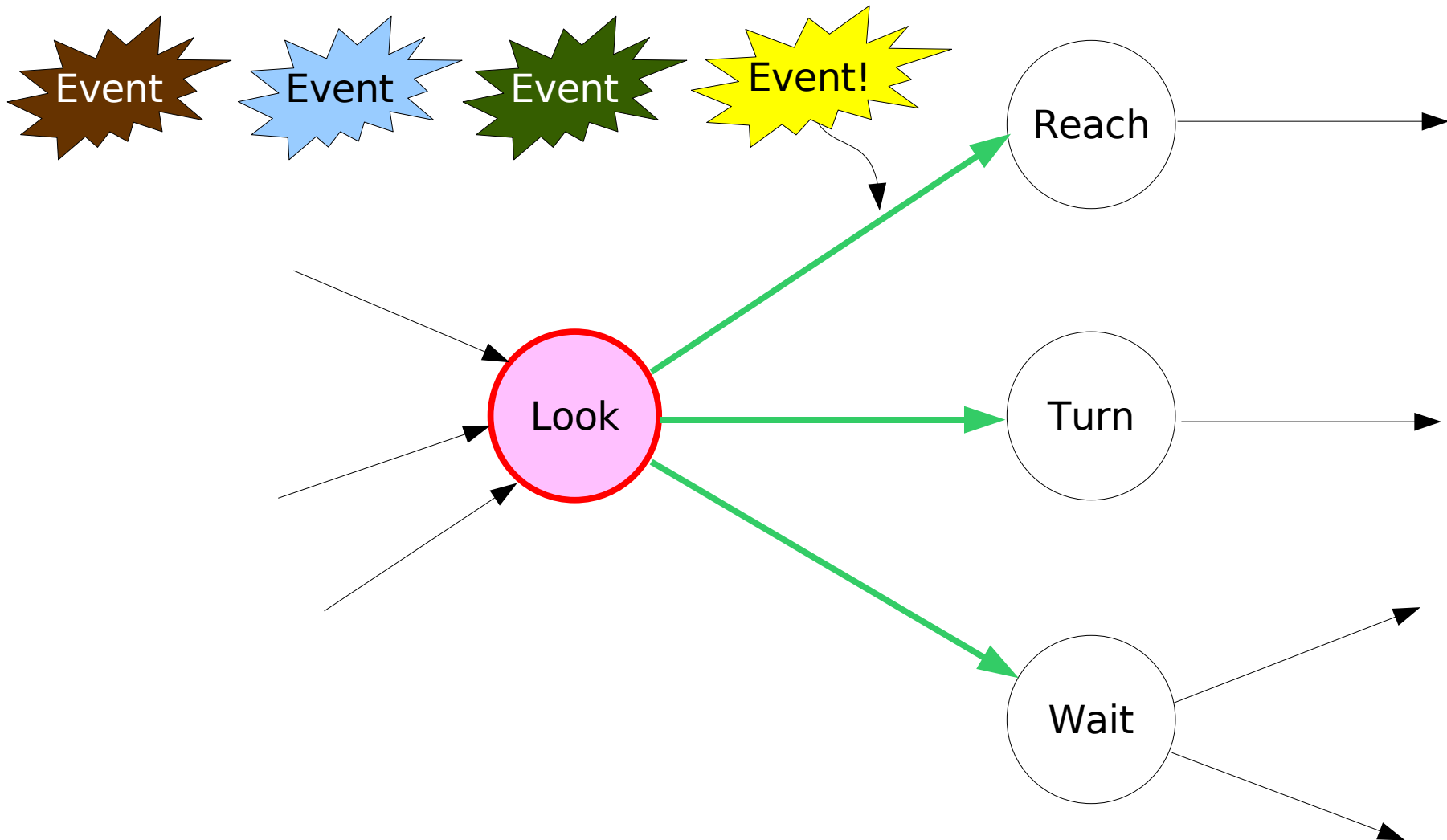
Outgoing transitions become active and begin listening for events.



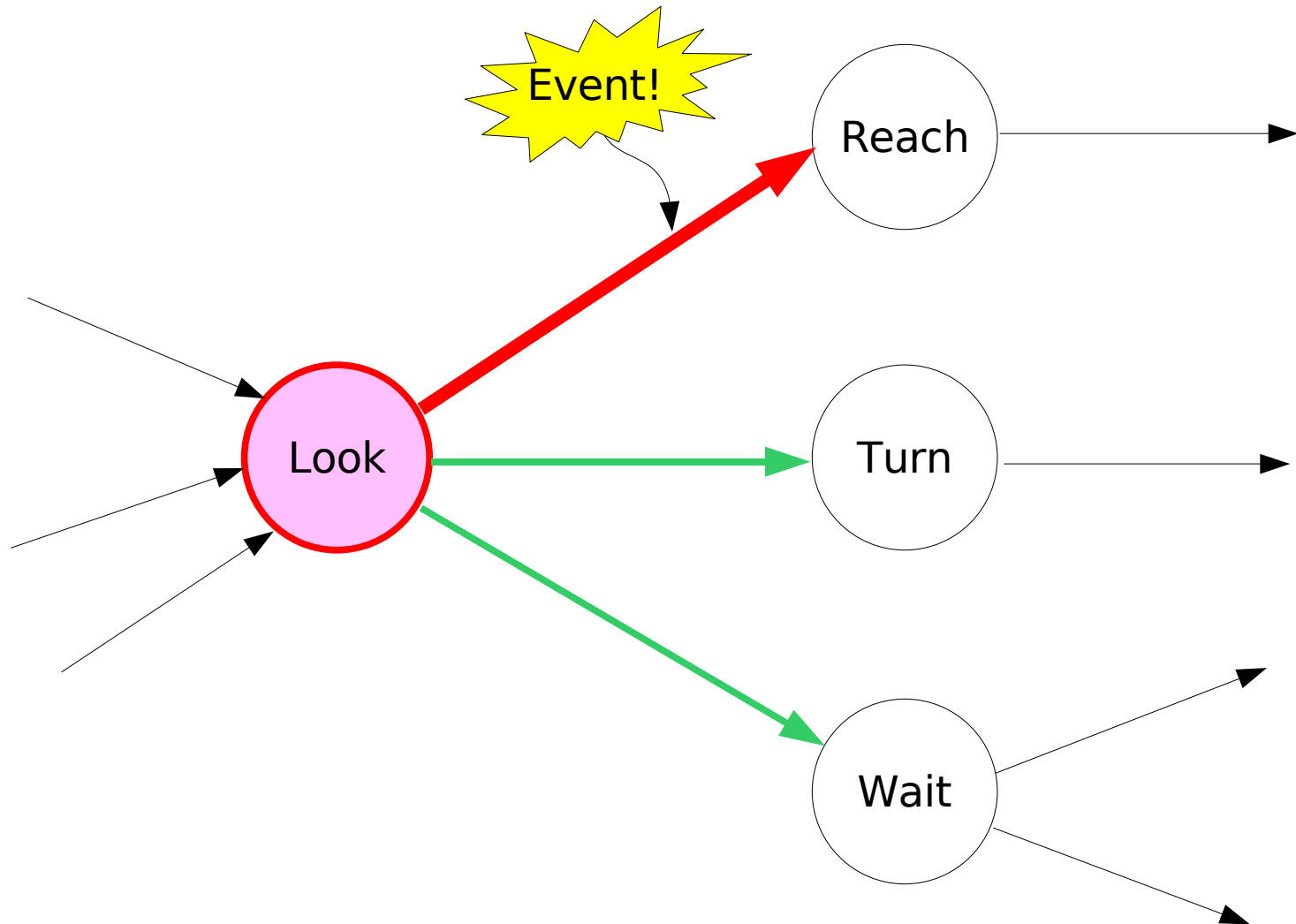
Random things happen....



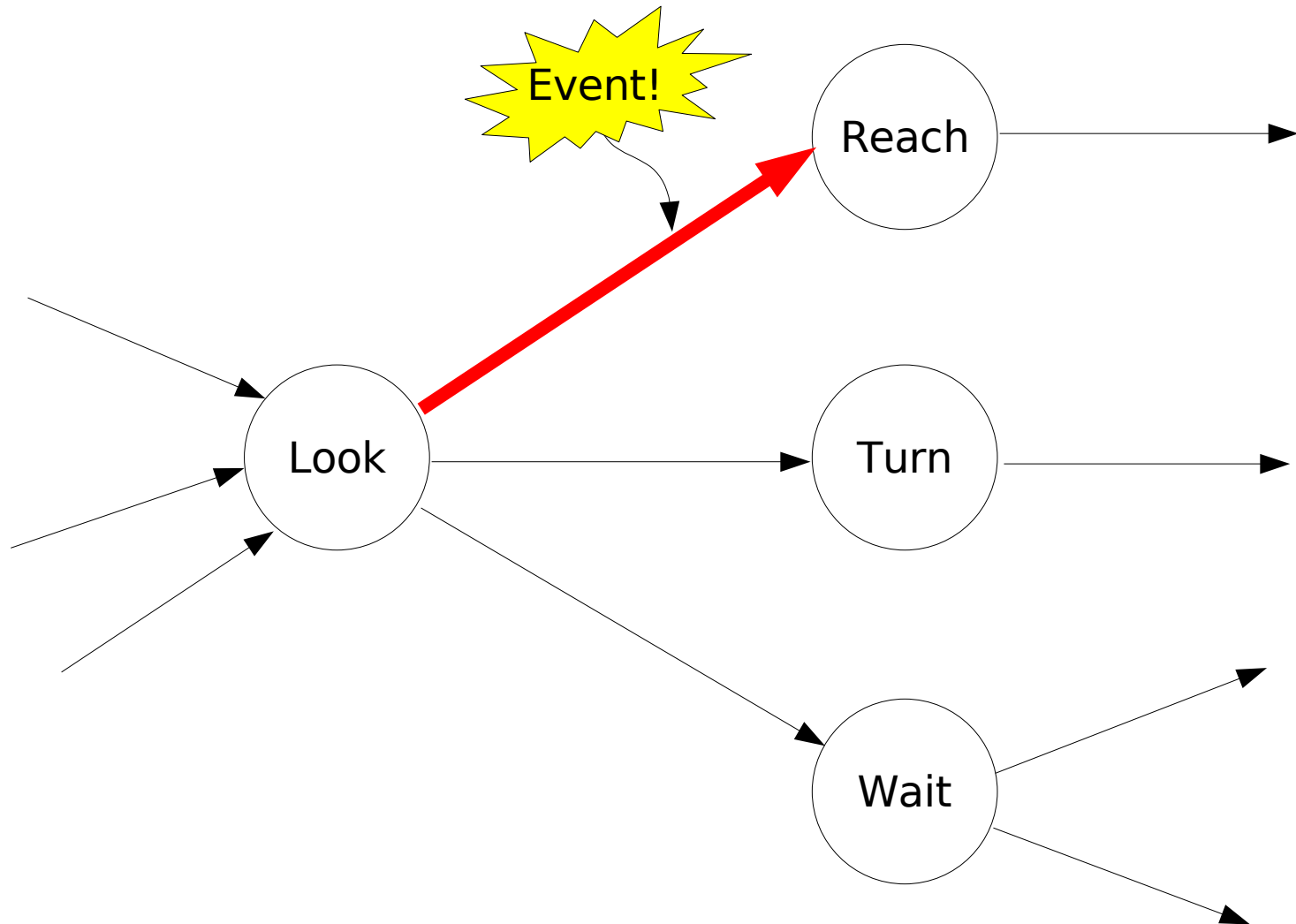
And then, something we've been looking for...



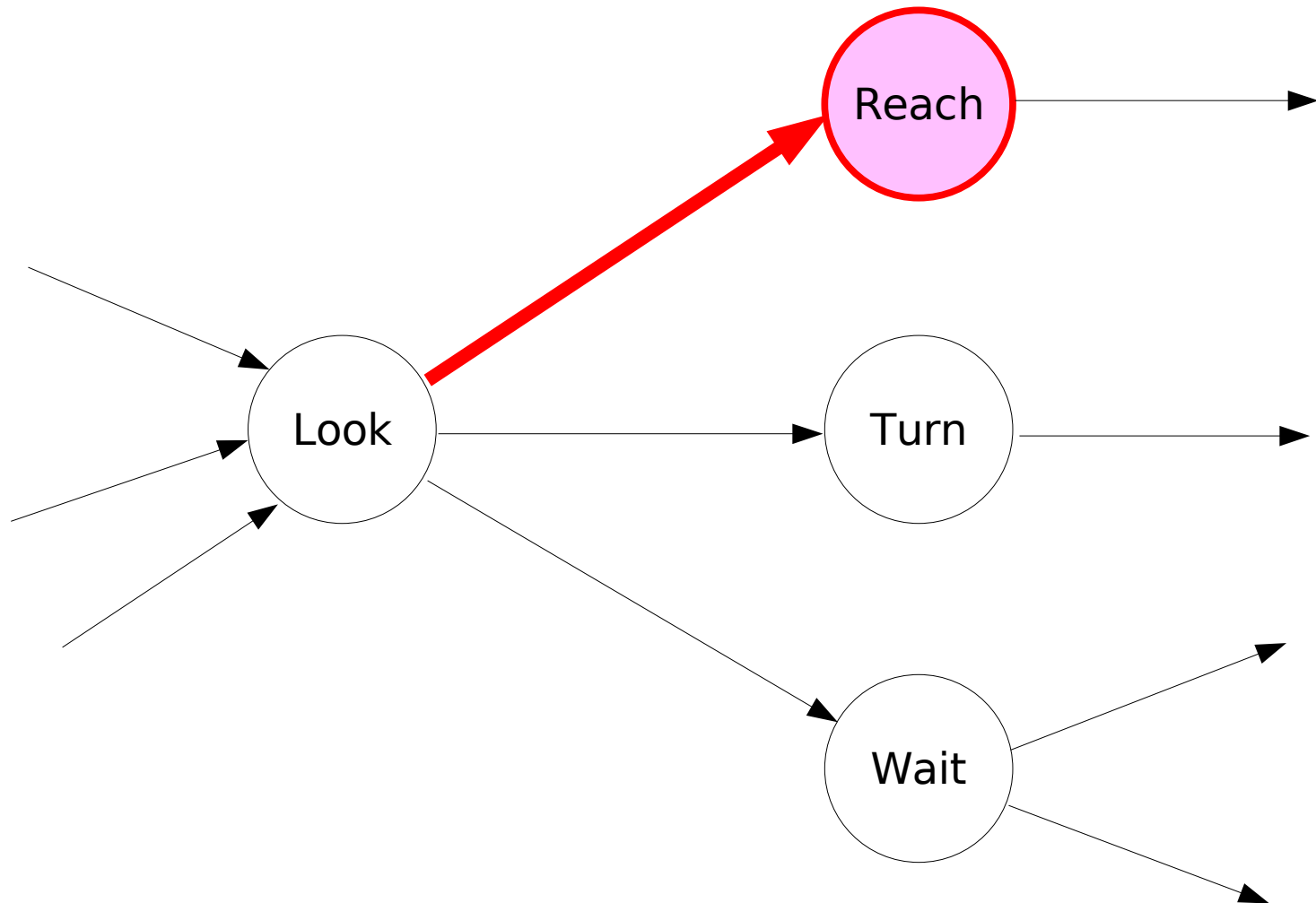
Transition decides to fire.



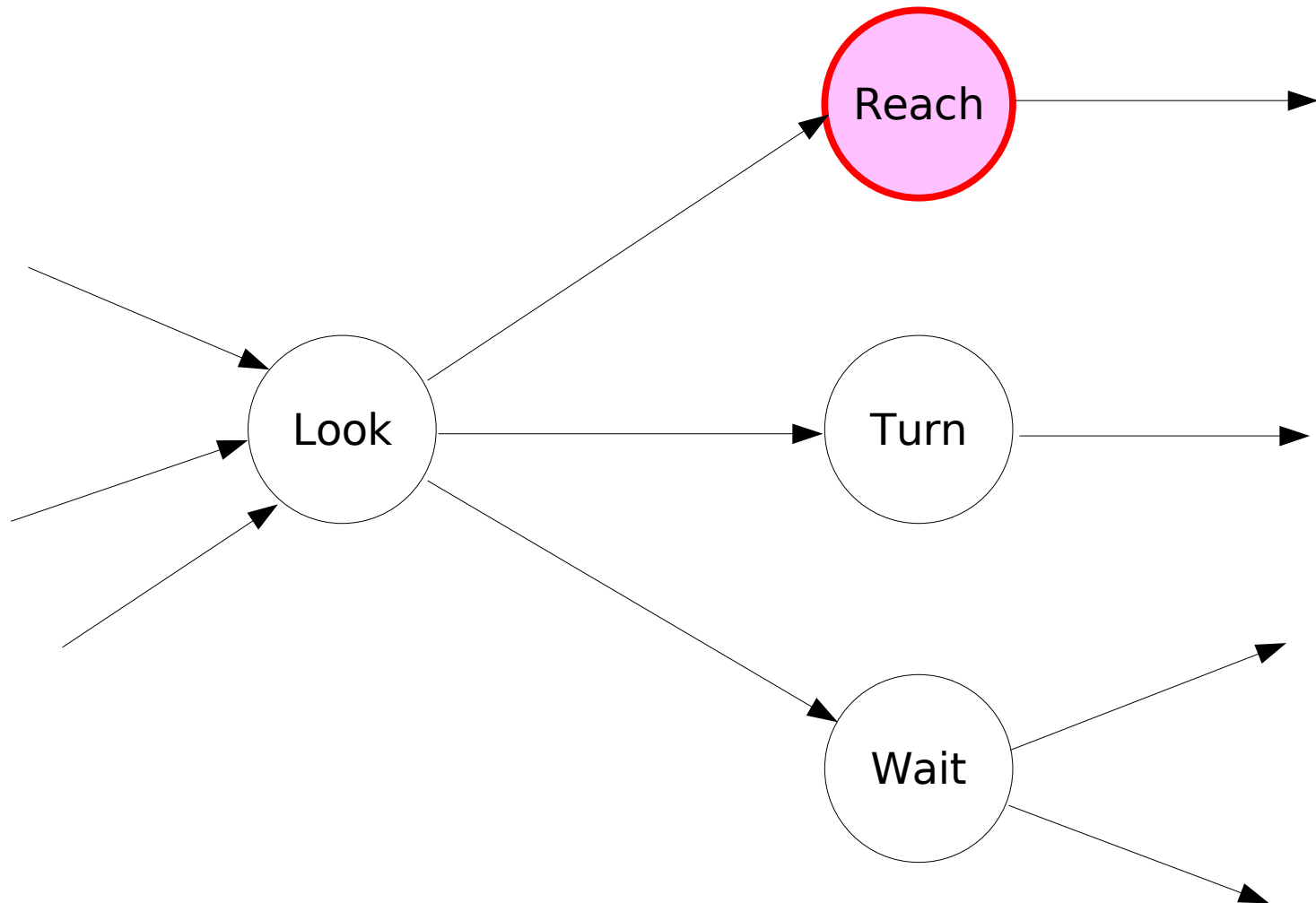
Transition deactivates the source node, Look.



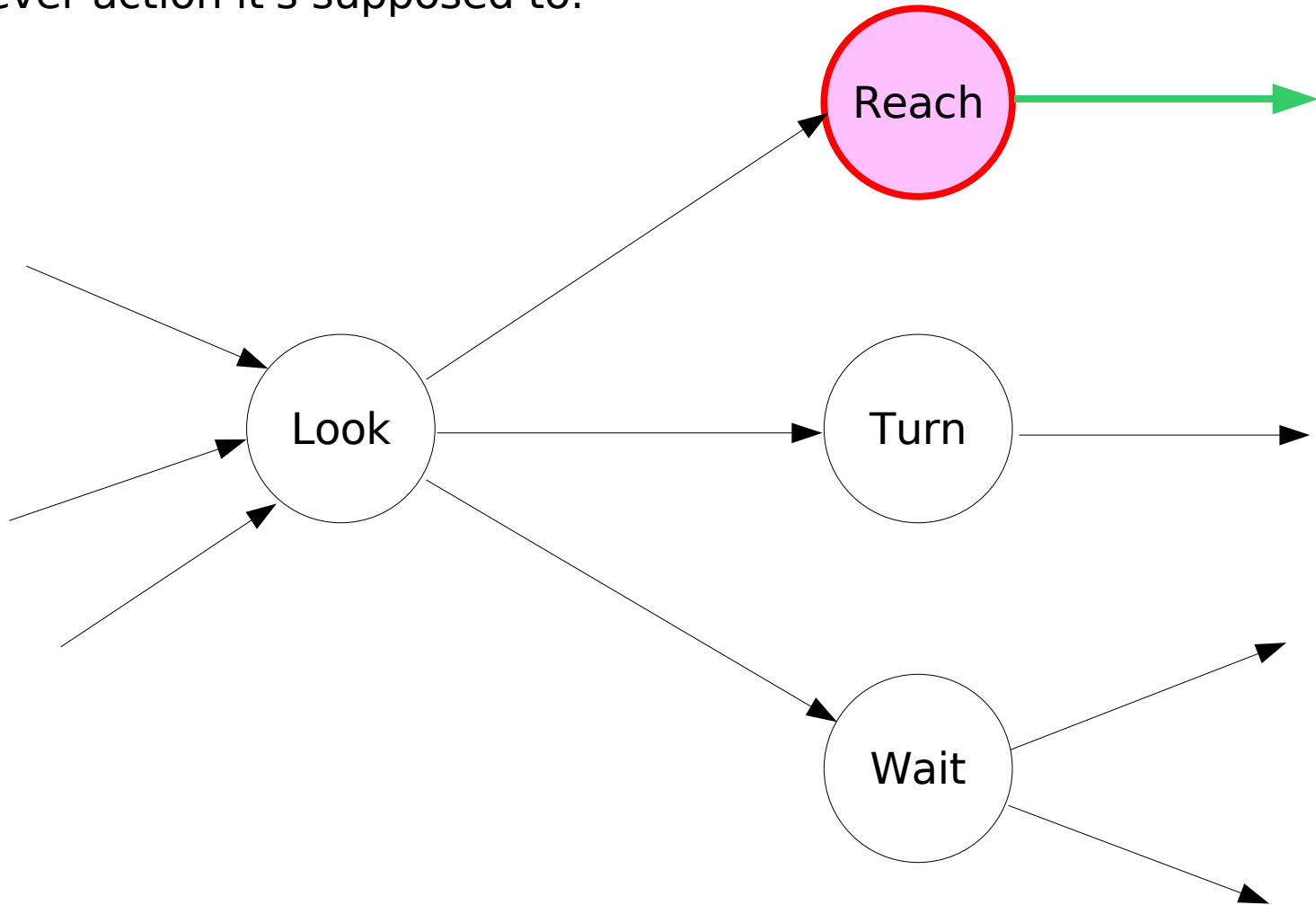
Transition activates the destination node, Reach.



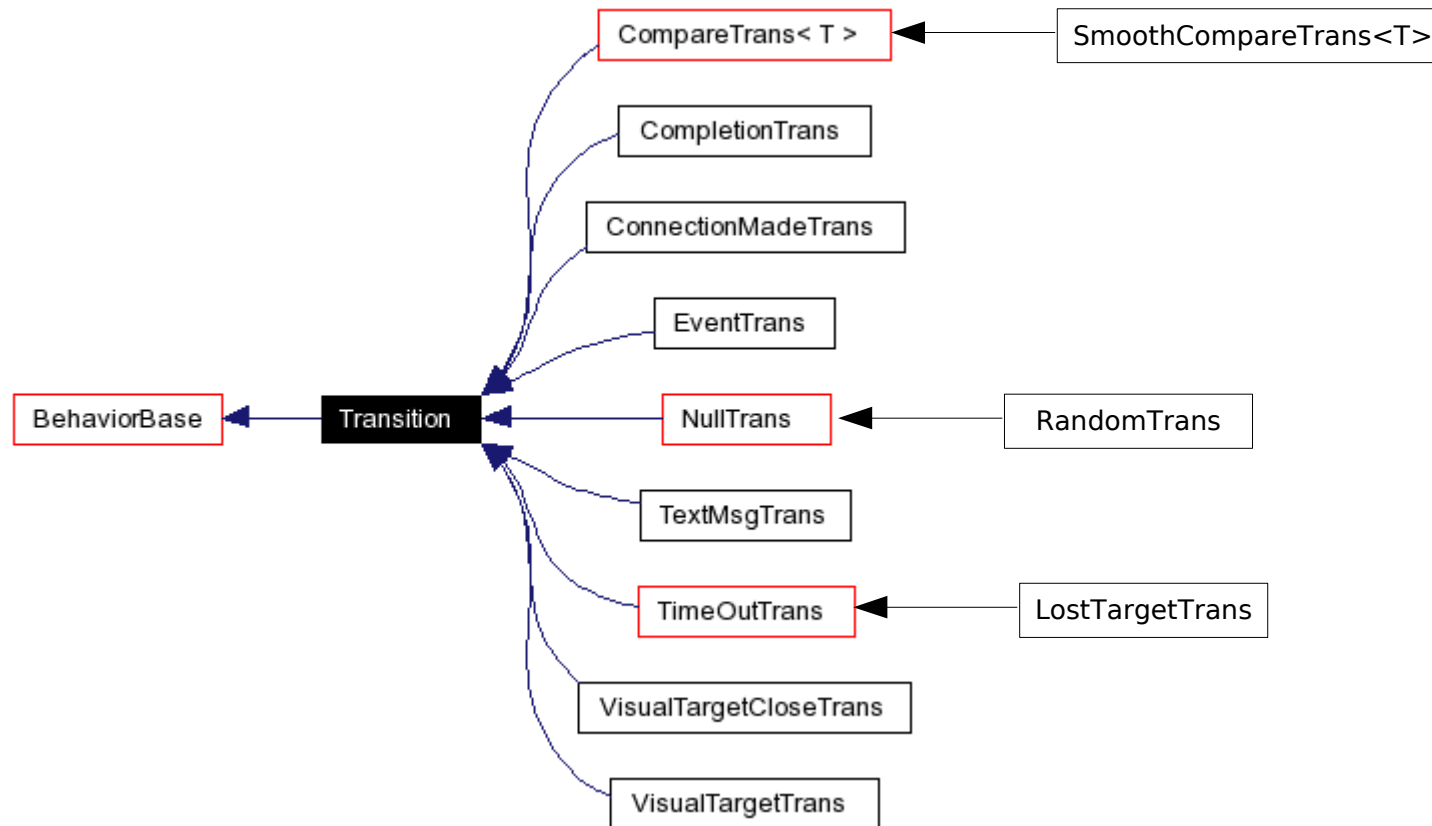
Transition deactivates.



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



Transition Types



State Machine Compiler

- Tekkotsu programmers don't normally write C++ code to build state machines one node or link at a time.
- They use a shorthand notation instead.
- The shorthand is turned into C++ by a state machine compiler.
- But to understand what the shorthand is doing, we need to build our first state machine by hand.



Programs As State Machines

Your program is the parent StateNode:

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

class BarkHowlBlinkBehavior : public StateNode {

public:
    BarkHowlBlinkBehavior() :
        StateNode("BarkHowlBlinkBehavior") {}
}
```

Setup and Teardown

- Programs must include a `setup()` function to construct the state machine as a child of the parent state node.
- `setup()` is called automatically the first time the parent's `start()` is called.
- A `teardown()` function is automatically provided to destroy the state machine. Called by `~StateNode()`.

Registering Nodes and Links

- Each node created by setup() must be registered with its parent using the addNode() method.

```
SoundNode *bark_node = new SoundNode("bark", "barkmed.wav");  
addNode(bark_node);
```

- Transitions are registered with their source nodes via the source node's addTransition() method.

```
bark_node->addTransition(new TimeOutTrans(howl_node, 5000));
```

- The variable startnode must be set to point to the starting node of the state machine.

Setup Example

```
virtual void setup() {
```

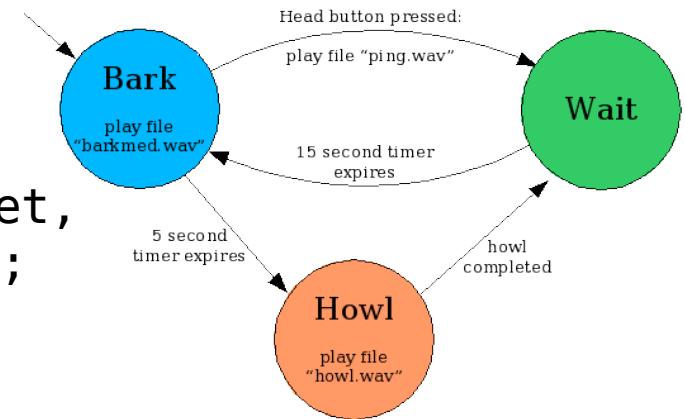
```
    SoundNode *bark_node = new SoundNode("bark", "barkmed.wav");  
    SoundNode *howl_node = new SoundNode("howl", "howl.wav");  
    StateNode *wait_node = new StateNode("wait");  
    addNode(bark_node); addNode(howl_node); addNode(wait_node);
```

```
    EventTrans *btrans =  
        new EventTrans(wait_node,  
                        EventBase::buttonEGID,  
                        ChiaraInfo::GreenButOffset,  
                        EventBase::activateETID);  
    btrans->setSound("ping.wav");  
    bark_node->addTransition(btrans);
```

```
    howl_node->addTransition(new CompletionTrans(wait_node));  
    wait_node->addTransition(new TimeOutTrans(bark_node, 15000));
```

```
    startnode = bark_node;
```

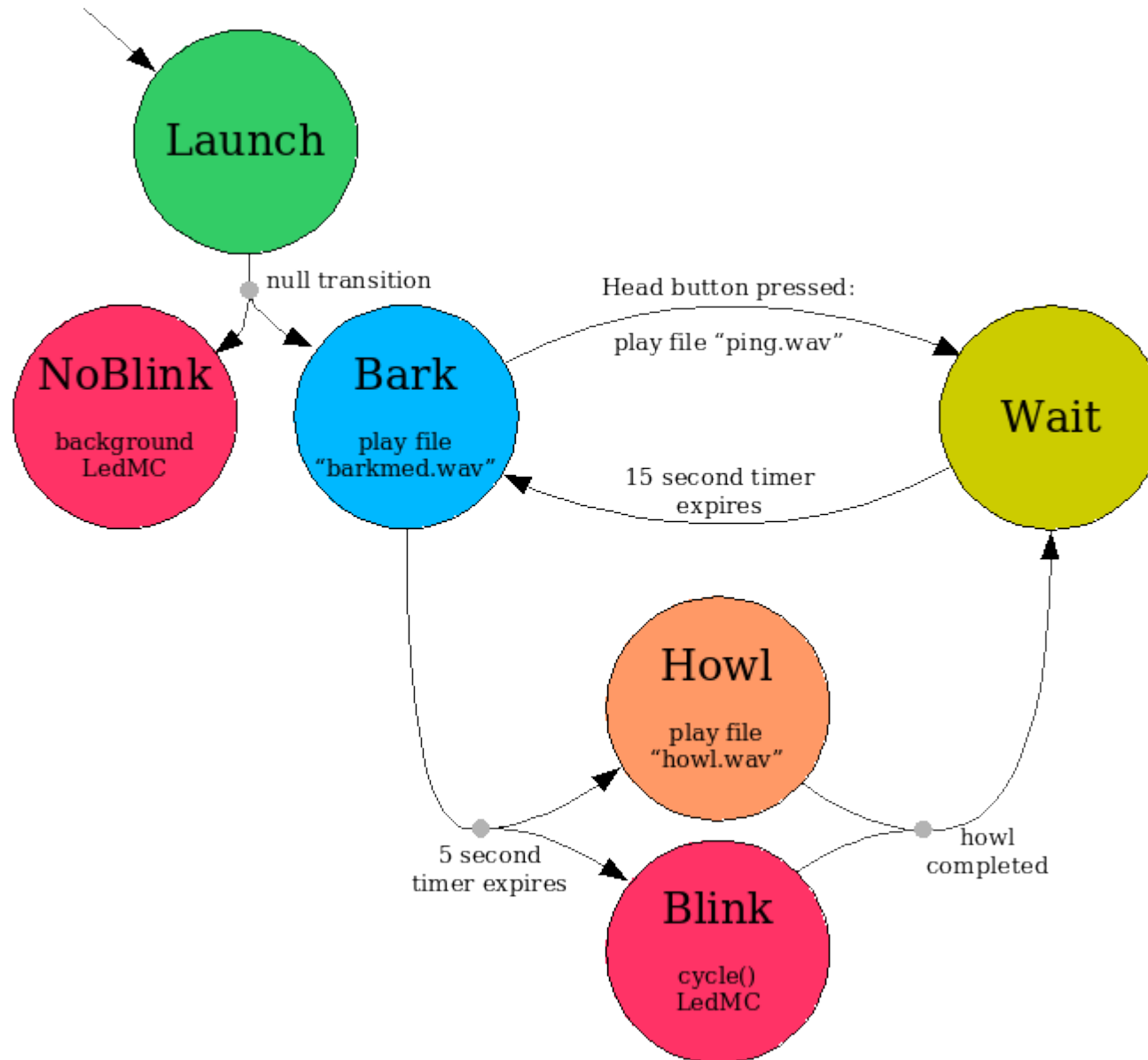
```
}
```



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()

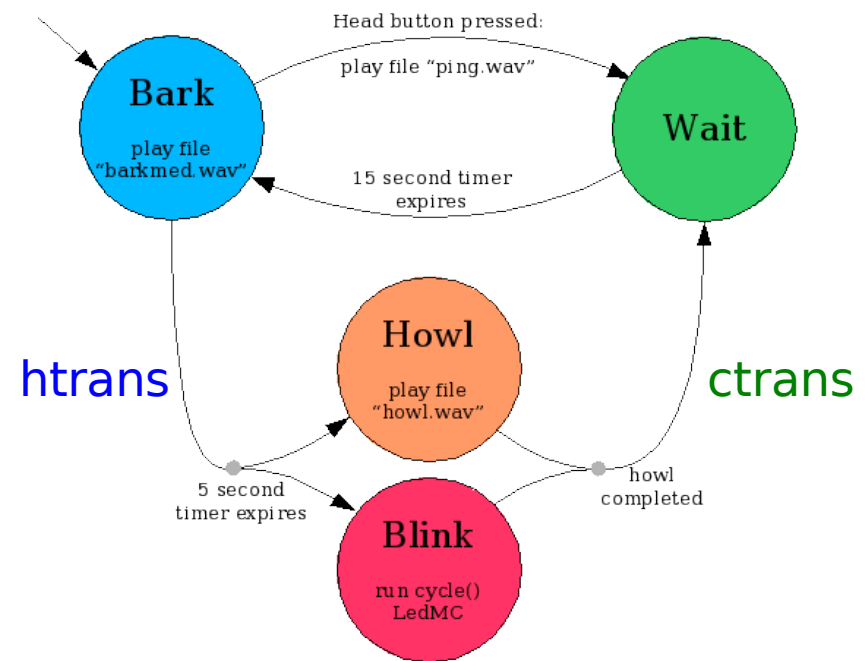
- Blink uses a motion command called LedMC, which is a child of LedEngine.
- The LedEngine::cycle() method never completes.
- When the howl completes, we want to leave both the howl state and the blink state.
- We can do this by telling CompletionTrans that only one of its source nodes needs to signal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

Setting Up the Blink

```
LedNode *blink_node = new LedNode("blink");  
addNode(blink_node);  
blink_node->getMC()->cycle(RobotInfo::AllLEDMask,1500,1.0);
```

```
TimeOutTrans *htrans = new TimeOutTrans(howl_node,5000);  
htrans->addDestination(blink_node);  
bark_node->addTransition(htrans);
```

```
CompletionTrans *ctrans = new CompletionTrans(wait_node,1);  
howl_node->addTransition(ctrans);  
blink_node->addTransition(ctrans);
```



Cleaning Up the Blink: Turn The LEDs Off

```
LedNode *noblink = new LedNode("noblink");
```

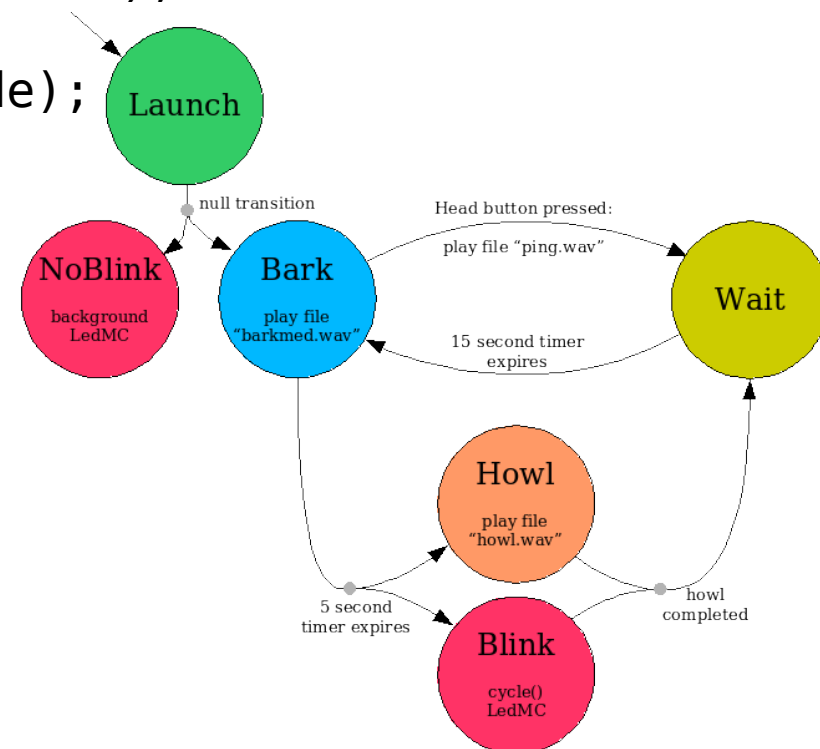
```
noblink->getMC()->set(RobotInfo::AllLEDMask, 0.0);  
noblink->setPriority(MotionManager::kBackgroundPriority);
```

```
StateNode *launcher = new Statenode("launcher");
```

```
NullTrans *ntrans = new NullTrans(bark_node);  
ntrans->addDestination(noblink);
```

```
launcher->addTransition(ntrans);
```

```
startnode = launcher;
```



Shorthand Notation

bark: SoundNode(\$,"barkmed.wav")

howl: SoundNode(\$,"howl.wav")

wait: StateNode

bark =T(5000)=> howl

bark =B(RobotInfo::GreenButOffset)=> wait

Shorthand Notation

- Node definition:

nodename: NodeClass(constructor_args)[initializers]

- 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:

source >==Transition==> {targ1name, targ2name, ...}

\$ and \$\$

- Use *\$* to refer to the name of the current node, e.g., these are equivalent:

foo: Statenode

foo: StateNode(\$)

foo: StateNode("foo")

bar: SoundNode(\$,"howl.wav")

bar: SoundNode("bar","howl.wav")

Must be present
to allow second
argument

- In long form, use *\$\$* to refer to the destination node of a transition, e.g., these are equivalent:

foo >==EventTrans(\$\$,EventBase::buttonEGID)==> bar

foo >==EventTrans(bar,EventBase::buttonEGID)==> bar

More Shorthand

>==NullTrans==>

$=N=>$

$\text{>==CompletionTrans==>}$

$=C=>$

$\text{>==CompletionTrans}(\$, \$ \$, n)\text{==>}$

$=C(n)=>$

$\text{>==TimeoutTrans}(\$, \$ \$, t)\text{==>}$

$=T(t)=>$

$\text{>==EventTrans}(\$, \$ \$, g, s, t)\text{==>}$

$=E(g, s, t)=>$

$\text{>== EventTrans}(\$, \$ \$, \text{EventBase::buttonEGID}, s)\text{==>}$

$=B(s)=>$

$\text{>== TextMsgTrans}(\$, \$ \$, \text{str})\text{==>}$

$=TM(\text{str})=>$

>==RandomTrans==>

$=RND=>$

$\text{>==SignalTrans}<T>(\$, \$ \$)\text{==>}$

$=S<T>=>$

$\text{>==SignalTrans}<T>(\$, \$ \$, v)\text{==>}$

$=S<T>(v)=>$

success or failure transitions

$=S=>$ or $=F=>$

```
virtual void setup() {  
  $statemachine{  
    startnode:StateNode =N=> {noblink, bark}  
  
    noblink: LedNode [setPriority(MotionManager::kBackgroundPriority);  
                      getMC()->set(RobotInfo::FaceLEDMask,0.0)]  
  
    bark: SoundNode($,"barkmed.wav")  
          =B(GreenButOffset)[setSound("ping.wav")]=> wait  
  
    wait: StateNode =T(15000)=> bark  
  
    bark =T(5000)=> {howl, blink}  
  
    howl: SoundNode($,"howl.wav")  
  
    blink: LedNode [getMC()->cycle(RobotInfo::AllLEDMask, 1500, 1.0)]  
  
    {howl, blink} =C(1)=> wait  
  }  
}  
} // end of setup()
```


Defining New Node Classes

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

$nodeclass MyMachine : StateNode {

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

    $nodeclass 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”.
- 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:

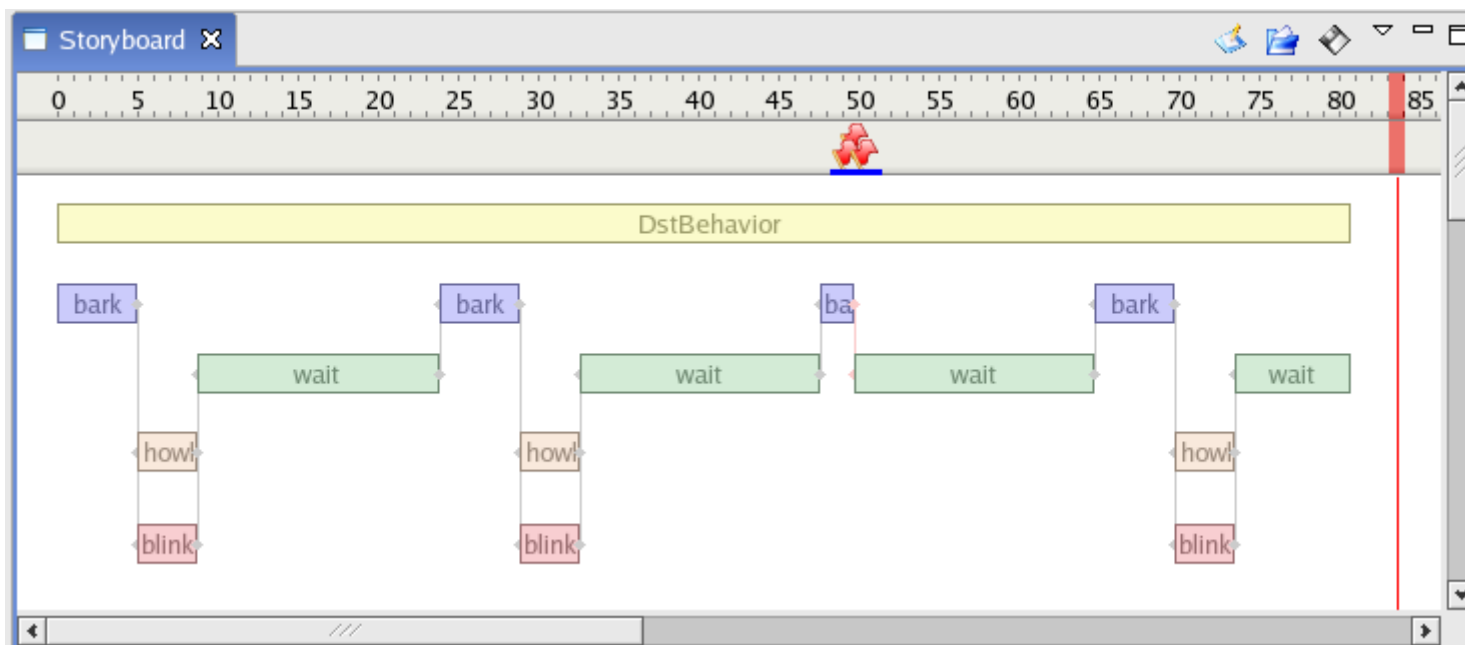
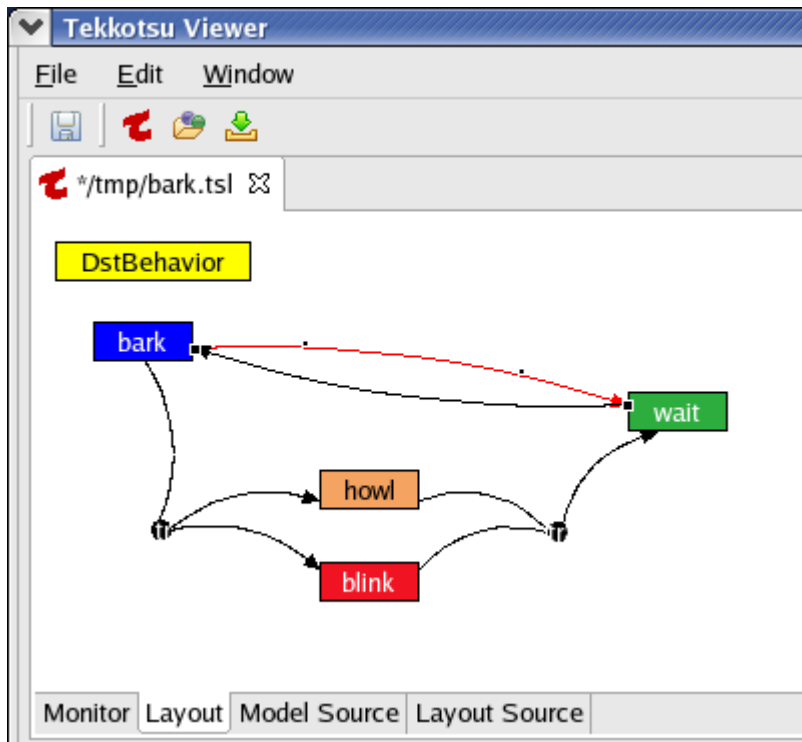
Tekkotsu/tools/sbin/stateparser BarkHowlBlinkBehavior.cc.fsm –

State Machine Events

- Entering or leaving a state generates a stateMachineEGID event.
 - activateETID for entering
 - deactivateETID for leaving
- Firing of a transition generates a stateTransitionEGID event.
- SignalTrans looks for a stateSignalEGID event
- You can use the Tekkotsu Event Logger to monitor these events:

Root Control > Status Reports > Event Logger

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 for "Explore State Machine". The model consists of several states (represented by ovals) and transitions (represented by arrows). The states include: Pink, Follow, Sit, Funny, Timer, Sound, Up, Punch, Look, Sniff, and Down. The transitions are labeled with actions like "Follow", "Sit", "Sound", "Up", "Punch", "Look", and "Sniff".

Below the model, there are tabs for "Monitor", "Layout", "Model Source", and "Layout Source". The "Monitor" tab is currently selected, showing a storyboard display. The storyboard is a timeline view of the state machine's execution. It features a horizontal axis with time markers from 0 to 60. A vertical red line indicates the current time, which is approximately 27.5 seconds. The storyboard shows the sequence of states and transitions as they occur over time. For example, the "Pink" state is active from 0 to 8.5 seconds, followed by "Follow" from 8.5 to 10 seconds, and "Sit" from 10 to 27.5 seconds. The "Timer" state is active from 10 to 27.5 seconds. The "Sound" state is active from 27.5 to 30 seconds, followed by "Up" from 30 to 35 seconds, "Punch" from 35 to 40 seconds, "Look" from 40 to 45 seconds, and "Sniff" from 45 to 50 seconds.

On the right side of the interface, there is a "Properties" panel. It shows the current selection (14.256s) and provides details about the selected state or transition. The selected state is "Sit", which is active at 27.002s and deactivates at 27.5s. The selected transition is "Timer--:Sit", which fires at 27.001s.

At the bottom of the interface, there is a "Storyboard" tab and an "Image Preview" window. The "Storyboard" tab shows a timeline view of the state machine's execution, with a vertical red line indicating the current time. The "Image Preview" window is currently empty.

Storyboard Tool: Snapshots

The screenshot displays the Tekkotsu Viewer application, which is used for monitoring and debugging 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 is a tabbed interface showing the current state machine configuration.
- Host Configuration:** Fields for Host (localhost), Port (10080), and Name (Explore State Machine). Buttons for Download Model, New Trace, and a refresh icon are also present.
- Storyboard View:** A timeline view showing the sequence of events. The timeline has a red vertical line indicating the current time. The storyboard shows a sequence of events: a yellow bar labeled "Logging Test" followed by a series of events (Image, Webcam, Message, Message, Image) and "Waiting" states.
- Properties Panel:** Displays the current selection (9.491s) and its properties. The selected event is "Image:Image", with a record at 8.457s and type: image. Below it, the "Waiting" state is shown with activate and deactivate times, and type: state. The "Logging Test" event is also listed with its activate and deactivate times.
- Image Preview:** A small window showing a live video feed from a webcam, displaying a room with a desk and a computer monitor.