

State Machines

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

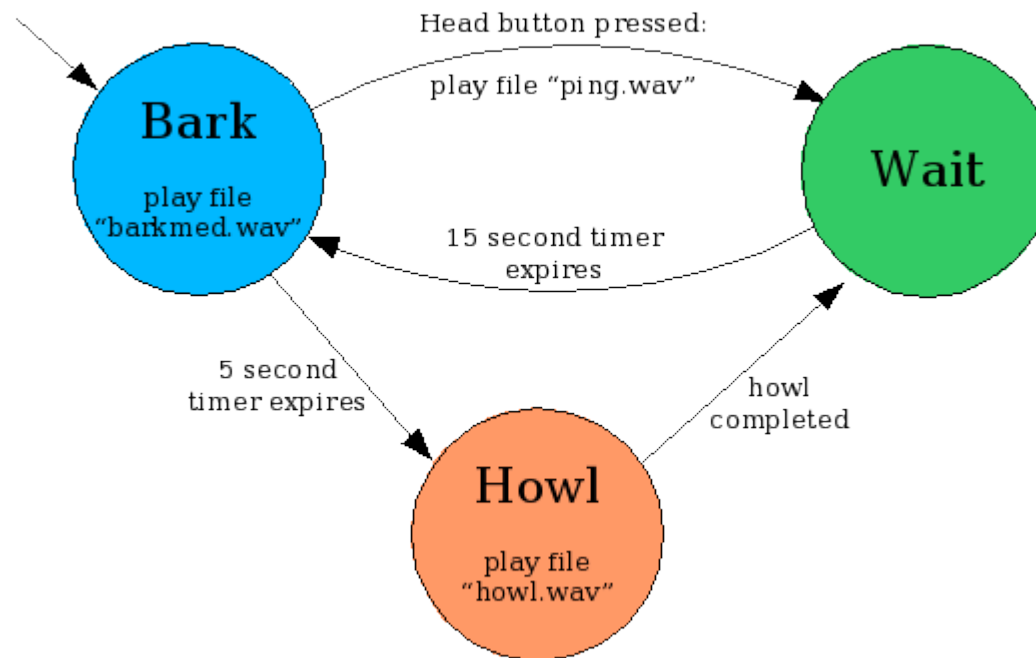
Carnegie Mellon
Spring 2008

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.



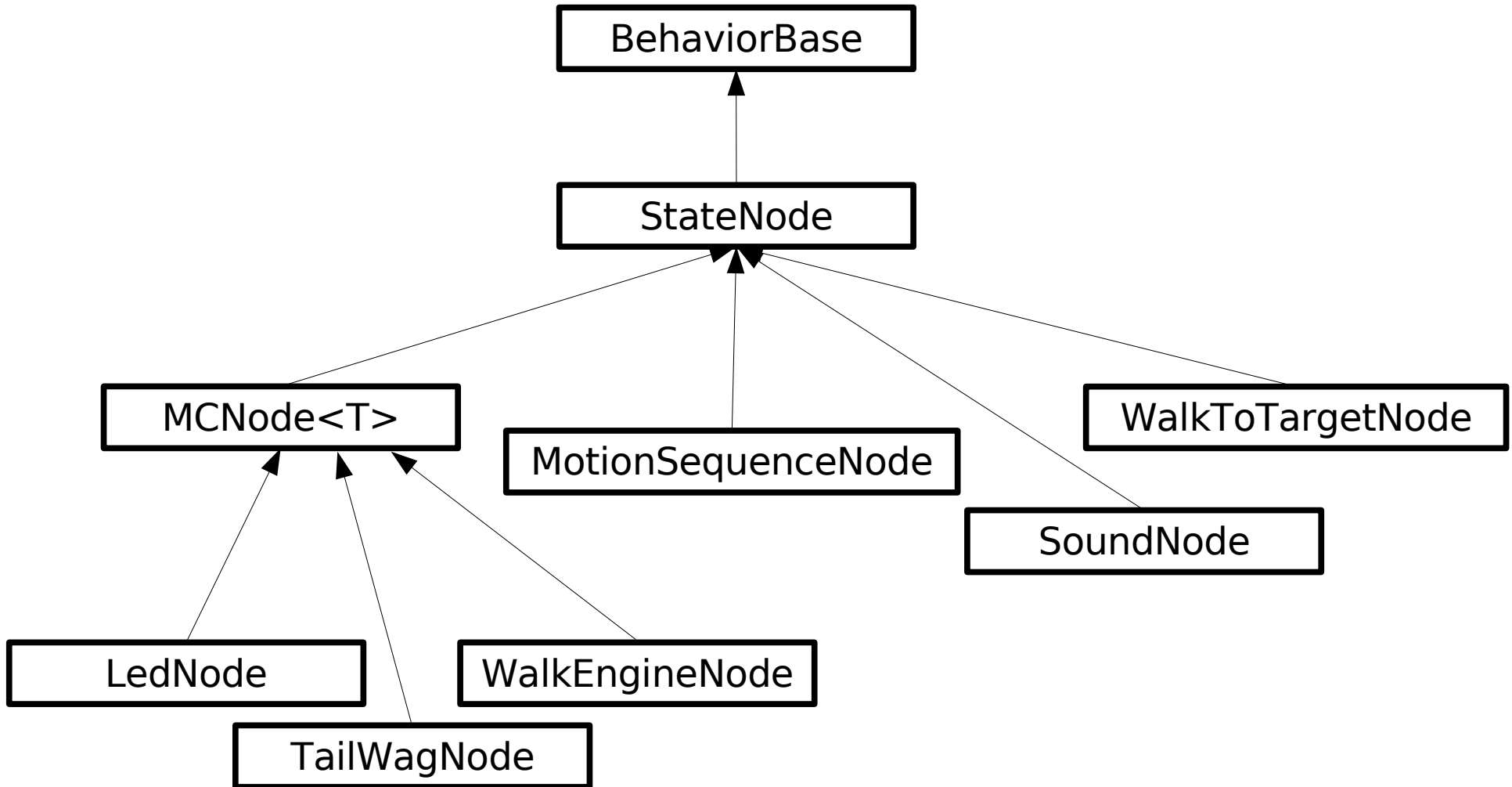
Extensions

- For convenience, we can extend the basic state machine idea to make programming easier.
- Extension 1: multi-states.
 - Several states can be active at once.
 - Provides for parallel processing.
- Extension 2: hierarchical structure.
 - State machines can nest inside other state machines.
 - Invocation sort of like a subroutine call.

Tekkotsu State Nodes

- In Tekkotsu, state machine nodes are *behaviors*.
- StateNode is a child of BehaviorBase.
- To enter a state, call its DoStart() method.
- To leave a state, call its DoStop() method.
- StateNodes can listen for and process events just like any other behavior.

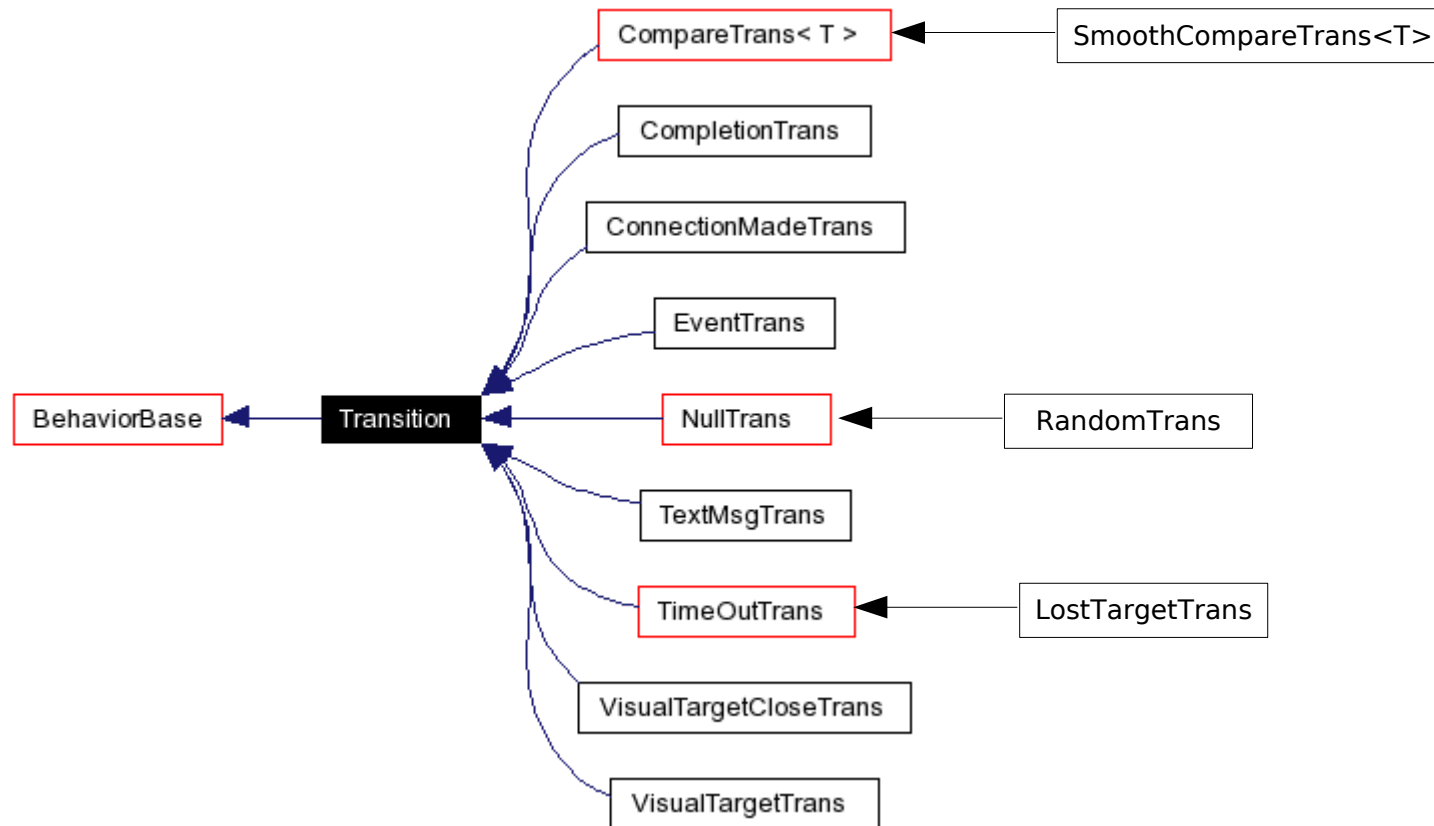
Types of State Nodes



Transitions

- Transitions in Tekkotsu are also *behaviors*.
 - Transition and StateNode are both subclasses of BehaviorBase.
- A transition's DoStart() 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 activates its target node(s).

Transition Types



Programs As State Machines

Your program is the parent StateNode:

```
#include "Behaviors/StateNode.h"
#include "Behaviors/Nodes/SoundNode.h"
#include "Behaviors/Transitions/CompletionTrans.h"
#include "Behaviors/Transitions/EventTrans.h"
#include "Behaviors/Transitions/TimeOutTrans.h"

class DstBehavior : public StateNode {

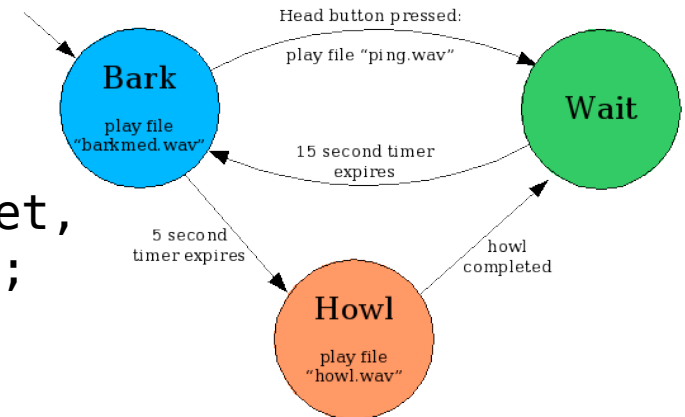
public:
    DstBehavior() : StateNode("DstBehavior") {}
```

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 `DoStart()` is called.
- Each node created by `setup()` must be registered with the parent using the `addNode()` method.
- Transitions are registered with their source nodes.
- A `teardown()` function is automatically provided to destroy the state machine. Called by `~StateNode()`.

Setup Example

```
virtual void setup() {  
    StateNode::setup();  
    cout << getName() << " setting up the state machine." << endl;  
  
    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,  
                        RobotInfo::HeadFrButOffset,  
                        EventBase::activateETID);  
    btrans->setSound("ping.wav");  
    bark_node->addTransition(btrans);  
  
    bark_node->addTransition(new TimeoutTrans(howl_node, 5000));  
    howl_node->addTransition(new CompletionTrans(wait_node));  
    wait_node->addTransition(new TimeoutTrans(bark_node, 15000));  
  
    startnode = bark_node;  
}
```



Parent's DoStart and DoStop

```
virtual void DoStart() {
    cout << getName() << " is starting up." << endl;
    StateNode::DoStart();
}

virtual void DoStop() {
    StateNode::DoStop();
    cout << getName() << " is shutting down." << endl;
}

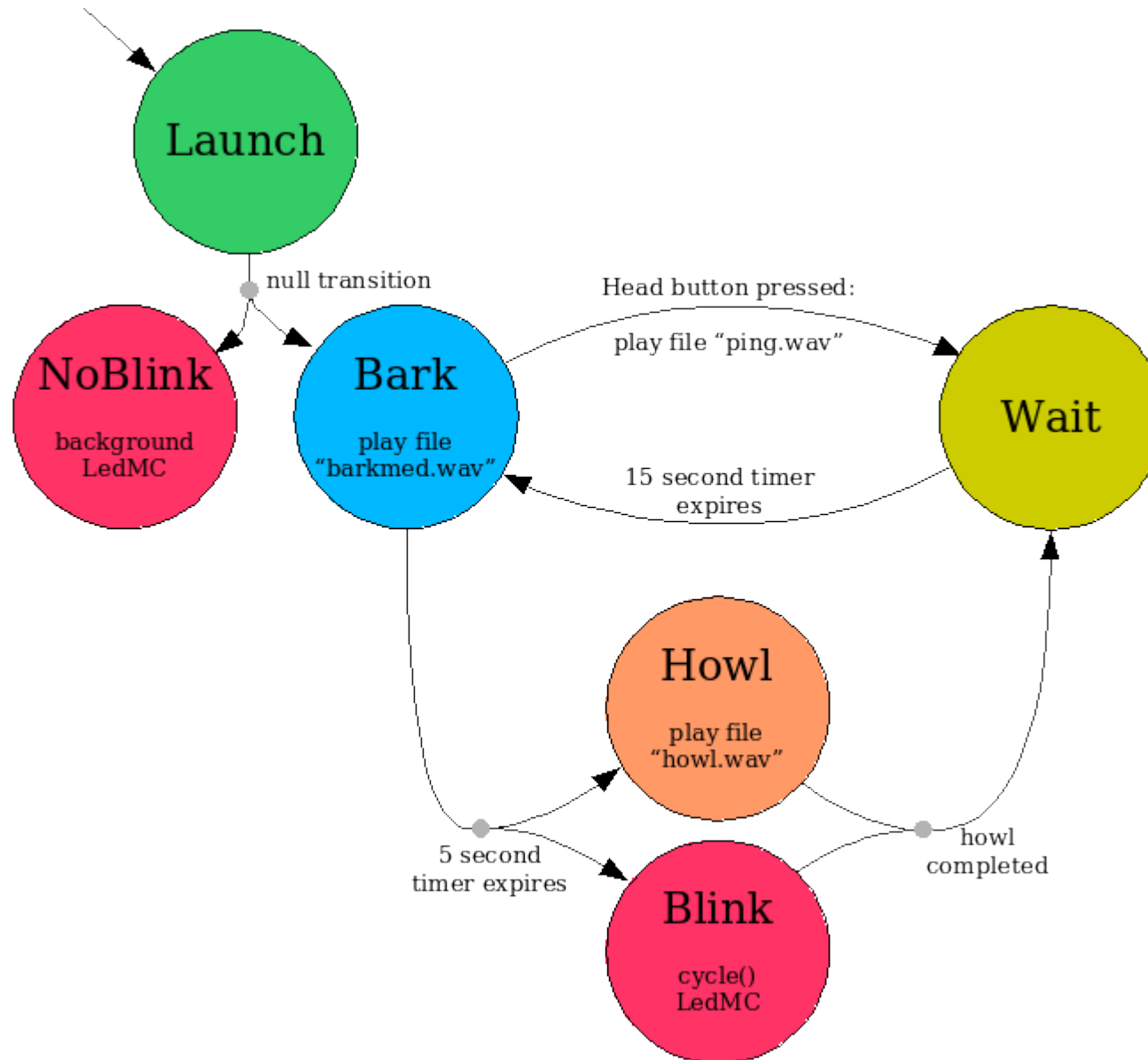
private: // Dummy functions to satisfy the compiler
    DstBehavior(const DstBehavior&);
    DstBehavior& operator=(const DstBehavior&);
```

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.
- You can use the Tekkotsu Event Logger to monitor these events:

Root Control > Status Reports > Event Logger

Multi-State Machines



Blink Using LedEngine::cycle()

- The cycle() motion command 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.

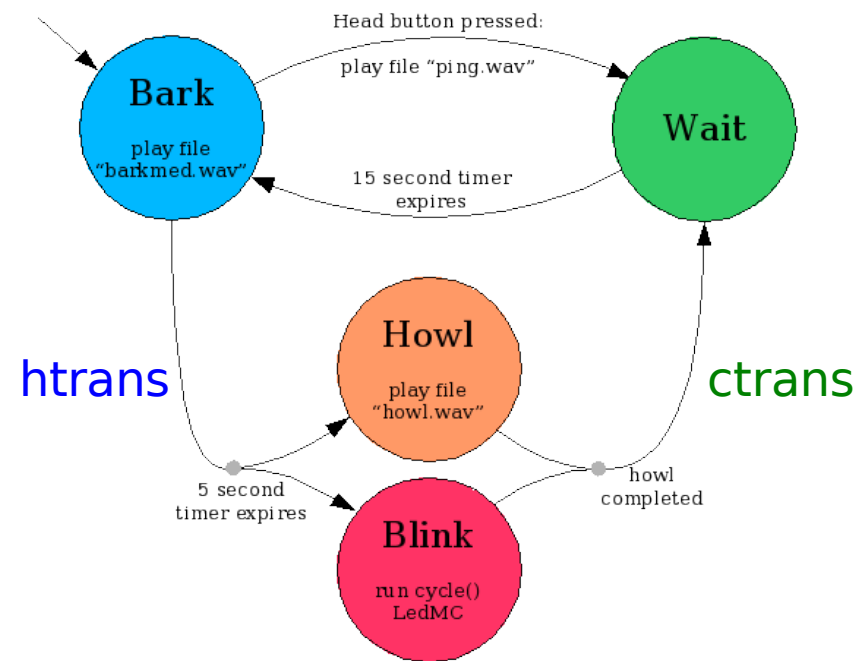
Setting Up the Blink

```
#include "Behaviors/Nodes/LedNode.h"
```

```
LedNode *blink_node = new LedNode("blink");  
addNode(blink_node);  
blink_node->getMC()->cycle(RobotInfo::FaceLEDMask,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 Face LEDs Off

```
LedNode *noblink = new LedNode("noblink");  
  
noblink->getMC()->set(RobotInfo::FaceLEDMask, 0.0);  
noblink->setPriority(MotionManager::kBackgroundPriority);  
  
StateNode *launcher = new Statenode("launcher");  
  
NullTrans *ntrans = new NullTrans(bark_node);  
ntrans->addDestination(noblink);  
  
launcher->addTransition(ntrans);
```

Shorthand Notation

- Node definition:

nodename: NodeClass(constructor_args)[initializers]

- Transition definition: source \Rightarrow Transition \Rightarrow target

sourcenode \Rightarrow transname:

TransitionClass(constructor_args)[initializers] \Rightarrow targetnode

- Multiple sources/targets:

source \Rightarrow Transition \Rightarrow {targ1name, targ2name, ...}

-

\$ and \$\$

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

foo: Statenode

foo: StateNode($\$$)

foo: StateNode("foo")

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

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

- 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=>
- CompletionTrans =C=>
- CompletionTrans(\$,\$\$,n) =C(n)=>
- TimeoutTrans(\$,\$\$,t) =T(t)=>
- EventTrans(\$,\$\$,g,s,t) =E(g,s,t)=>
- SignalTrans<T> =S<T>=>
- SignalTrans<T>(\$,\$\$,v) =S<T>(v)=>

Bark/Woof in Shorthand

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

```
wait: StateNode
```

```
bark >== EventTrans($, $$,  
                    EventBase::buttonEGID,  
                    RobotInfo::HeadFrButOffset,  
                    EventBase::activateETID) [setSound("ping.wav");]  
    ==> wait
```

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

```
// now define howl and its transitions
```

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

```
bark =T(500)=> howl =C=> wait
```

```
virtual void setup() {  
    StateNode::setup();  
}
```

file: DstBehavior.h.fsm

```
#statemachine
```

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

```
leds_off: LedNode [setPriority(MotionManager::kBackgroundPriority);  
                  getMC()->set(RobotInfo::FaceLEDMask,0.0);]
```

```
bark: SoundNode($,"barkmed.wav")  
    >== EventTrans($, $$,  
                  EventBase::buttonEGID,  
                  RobotInfo::HeadFrButOffset,  
                  EventBase::activateETID) [setSound("ping.wav");] ==>  
    wait: StateNode =T(15000)=> bark
```

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

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

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

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

```
#endstatemachine
```

```
startnode = launch; } // end of setup()
```

Creating New State Node Types

- Only a few types of state node classes are built in. You will probably need to write some new ones.
- Example: let's write `PlayNTimesNode` that plays a sound N times.
- This node class will inherit from `SoundNode`.
- First, let's see how we'll use the node.

“Annoying Dog” State Machine

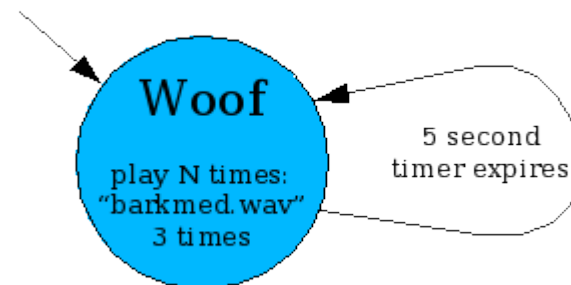
```
#include "PlayNTimesNode.h"

class DstBehavior : public StateNode {

public:
  DstBehavior() : StateNode("DstBehavior"), startnode(NULL) {}

  virtual void setup() {
    StateNode::setup();
    sndman->LoadFile("barkmed.wav");
#statemachine
    woof: PlayNTimesNode("$", "barkmed.wav", 3) =T(5000)=> woof
#endstatemachine
    startnode = woof;
  }

  virtual void teardown() {
    sndman->ReleaseFile("barkmed.wav");
    StateNode::teardown();
  }
}
```



PlayNTimesNode.h

```
#include "Behaviors/Nodes/SoundNode.h"

class PlayNTimesNode : public SoundNode {
protected:
    int ntimes;

public:
    PlayNTimesNode(std::string nodename="PlayNTimesNode",
                  std::string soundfilename="",
                  int _ntimes=1) :
        SoundNode("PlayNTimesNode", nodename, soundfilename),
        ntimes(_ntimes) {}

    void setNTimes(int const n) { ntimes = n; }

    virtual void DoStart() {
        SoundNode::DoStart();
        for (int i=2; i<=ntimes; i++)
            sndman->chainFile(curplay_id, filename);
    };
};
```

Inherited from SoundNode



PlayNTimesNode.h

protected:

```
PlayNTimesNode(std::string &classname,  
               std::string &nodename,  
               std::string &soundfilename,  
               int _ntimes=1) :  
    SoundNode(classname, nodename, soundfilename),  
    ntimes(_ntimes) {}
```

- This second form of the constructor is used for nodes that want to inherit from PlayNTimesNode.
- It's protected so that only subclasses can access it.

New Transition Types

- Transitions can maintain their own internal state and do complicated things, such as counting events.
- Example: let's define LostTargetTrans that fires if we lose sight of the pink ball for n seconds.
- To avoid being fooled by noise, the transition will require that the ball be seen for 5 camera frames in a row before resetting the timer.
- Transitions can have optional names, so we need three forms of constructor.

LostTargetTrans.h

```
class LostTargetTrans : public TimeOutTrans {
public:

    LostTargetTrans(StateNode* destination,
                    unsigned int source_id,
                    unsigned int timeLimit,
                    int minframes=5) :
        TimeOutTrans("LostTargetTrans", "LostTargetTrans",
                    destination, timeLimit),
        sid(source_id), minf(minframes), counter(0) {}

    LostTargetTrans(const string &name, StateNode* destination,
                    unsigned int source_id,
                    unsigned int delay, int minframes=5) :
        TimeOutTrans("LostTargetTrans", name, destination, timeLimit),
        sid(source_id), minf(minframes), counter(0) {}
};
```

LostTargetTrans.h

protected:

```
LostTargetTrans(const string &classname,  
                const string &instname,  
                StateNode* destination,  
                unsigned int source_id,  
                unsigned int timeLimit,  
                int minframes=5) :  
    TimeOutTrans(classname,instname,destination,timeLimit),  
    sid(source_id), minf(minframes), counter(0) {}
```

private:

```
unsigned int sid;  
int minf;    // # frames target must be seen before resetting timer  
int counter; // # frames target has been seen so far
```

LostTargetTrans.h

```
virtual void DoStart() {
    TimeoutTrans::DoStart();
    erouter->addListener(this,EventBase::visObjEGID,sid);
}

virtual void processEvent(const EventBase &e) {
    if (e.getGeneratorID()==EventBase::visObjEGID)
        if (e.getTypeID() != EventBase::deactivateETID) {
            ++counter;
            if (counter > minf) resetTimer();
        }
    else // must be a timer event
        TimeoutTrans::processEvent(e); // parent will call fire()
}

virtual void resetTimer() {
    TimeoutTrans::resetTimer();
    counter = 0;
}
```

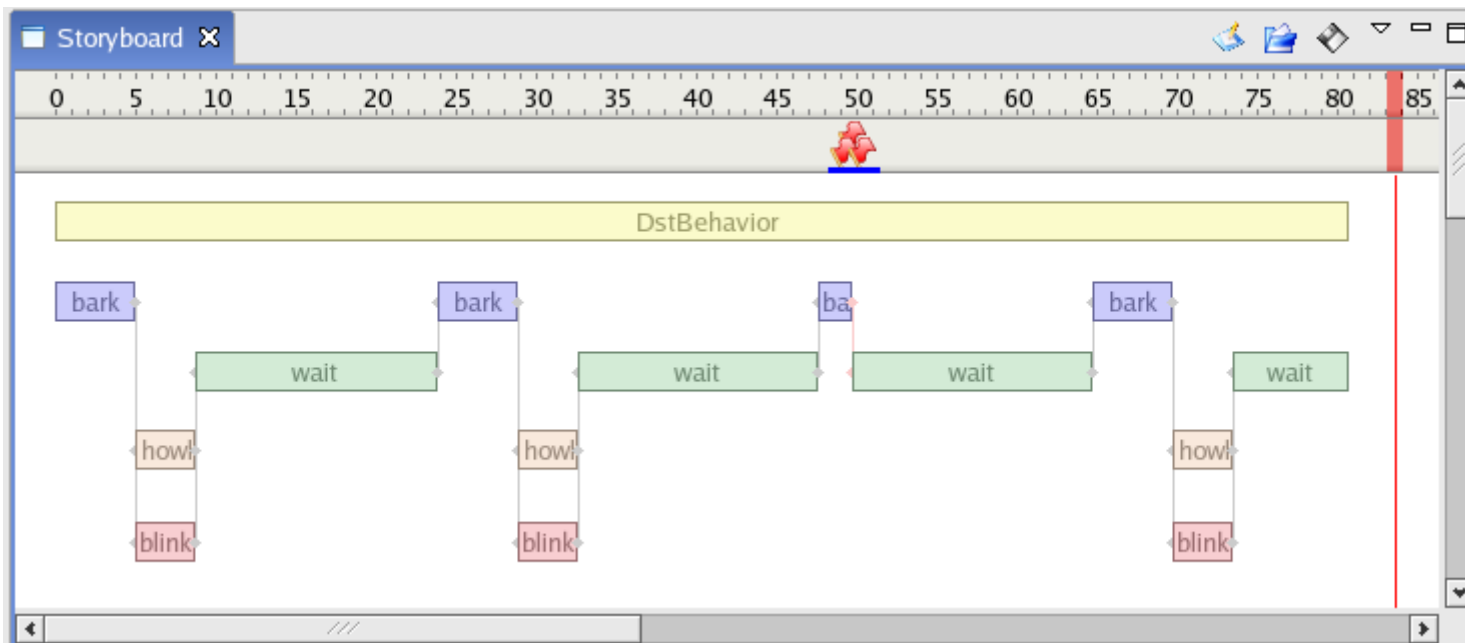
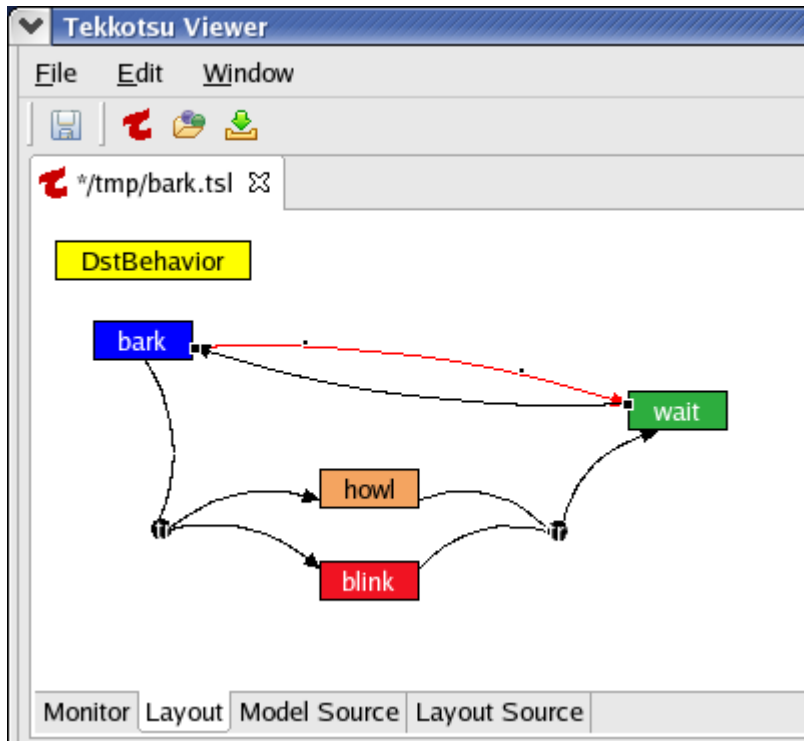
Other Transition Types

- NullTrans fires immediately.
 - Useful for nodes that just initiate an action and then move on.
- RandomTrans enters one of its target states at random.
- CompareTrans compares a memory location with a value, and fires if the specified test is met. For example, to transition when IR indicates 200 mm from an obstacle:

```
CompareTrans(node37,  
             &state->sensors[NearIRDistOffset],  
             CompareTrans::LT,  
             200,  
             EventBase(EventBase::sensorEGID,  
                       SensorSourceID::UpdatedSID,  
                       EventBase::statusETID))
```

- SignalTrans looks for a specified DataEvent
 - Useful for implementing “switch” statements

Storyboard Tool: State Machine Layout



Storyboard Tool: Storyboard Display

The screenshot displays the Tekkotsu Viewer interface. At the top, the title bar reads "Tekkotsu Viewer" with standard window controls. Below it is a menu bar with "File", "Edit", and "Window". The main workspace is divided into several sections:

- Model View:** Shows a state machine diagram with nodes like "Pink", "Follow", "Sit", "Funny", "Timer", "Sound", "Up", "Down", "Sniff", "Look", and "Punch".
- Properties Panel:** Located on the right, it shows "Current selection : 14.256s" and details for the selected state: "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).
- Storyboard:** A timeline at the bottom shows the execution sequence of the state machine. A red vertical line indicates the current time, which is approximately 27.5 seconds. The storyboard shows states like "Funny", "Timer", "Sit", "Sound", "Down", "Sniff", "Up", "Punch", and "Look" plotted against time.
- Image Preview:** A panel on the right side, currently empty, used for displaying visual output from the model.

At the bottom of the window, there is a status bar with the text "10/25/00" on the left and "15-454 Cognitive Robotics" in the center.

Storyboard Tool: Snapshots

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

- Top Panel:** Contains the menu bar (File, Edit, Window) and a toolbar with icons for file operations. Below this is a browser-like address bar showing the file path: `*/afs/cs.cmu.edu/user/dst/S...`.
- Host Configuration:** A section for setting the host and port. The host is set to `localhost` and the port to `10080`. The name of the state machine is `Explore State Machine`. Buttons for `Download Model`, `New Trace`, and a refresh icon are also present.
- Storyboard View (Bottom):** A timeline-based view showing the execution of the state machine. A yellow bar at the top represents the `Logging Test` state, which is active from time 0 to 45.41. Below this, 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 corresponding state in the top view by a vertical line. A red vertical line indicates the current time, which is approximately 8.510.
- Properties Panel (Right):** Shows the current selection at `:9.491s`. It lists the following state transitions:
 - `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.