# State Machines

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

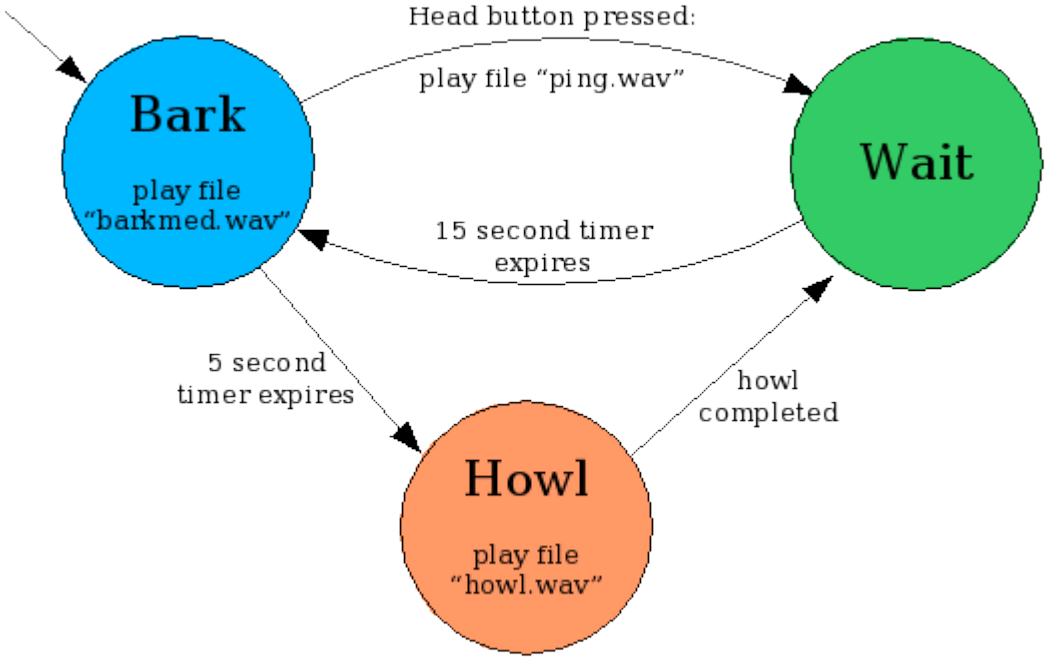
Carnegie Mellon
Spring 2007

# 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.



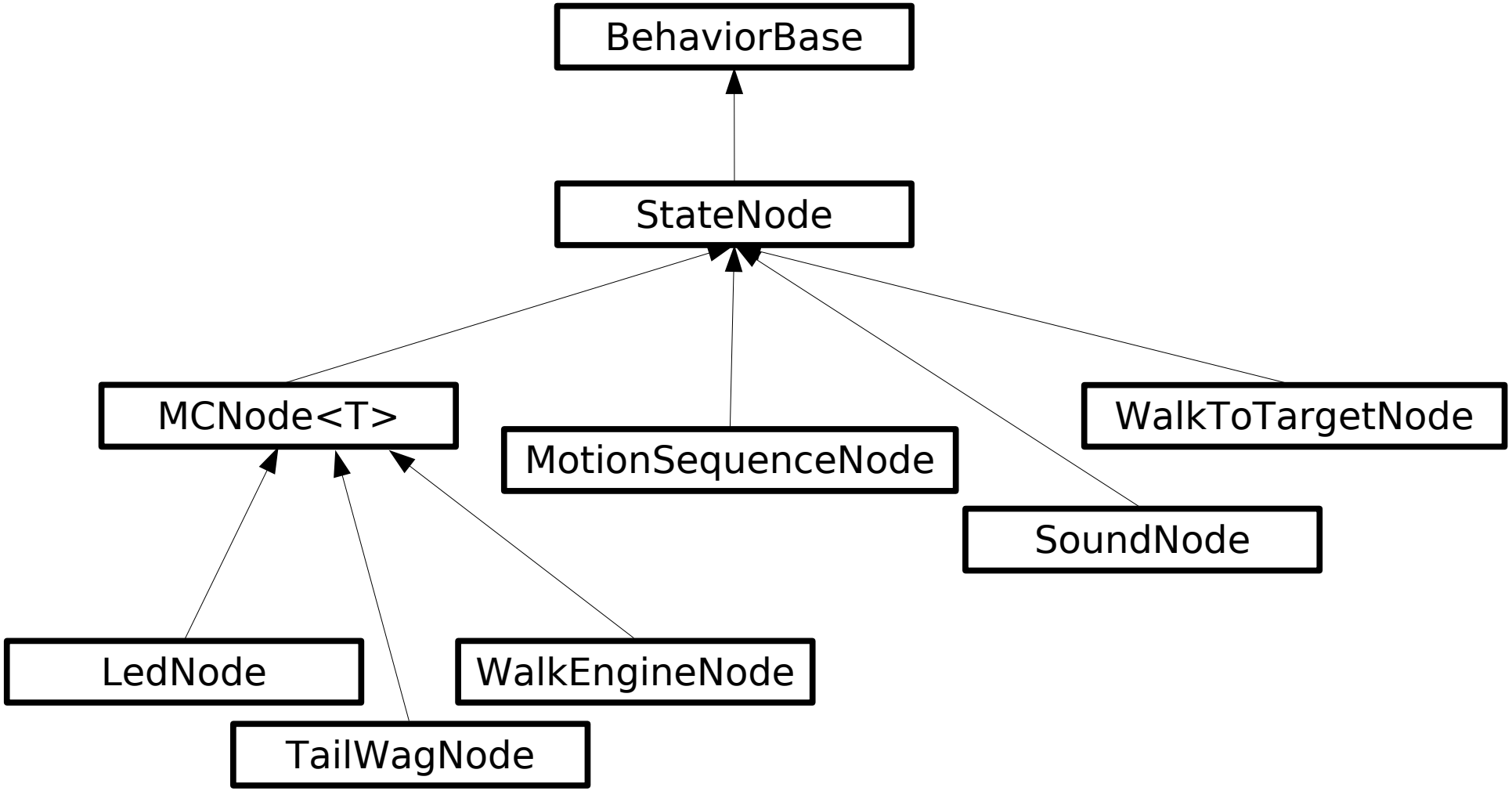
# Extensions

- 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.

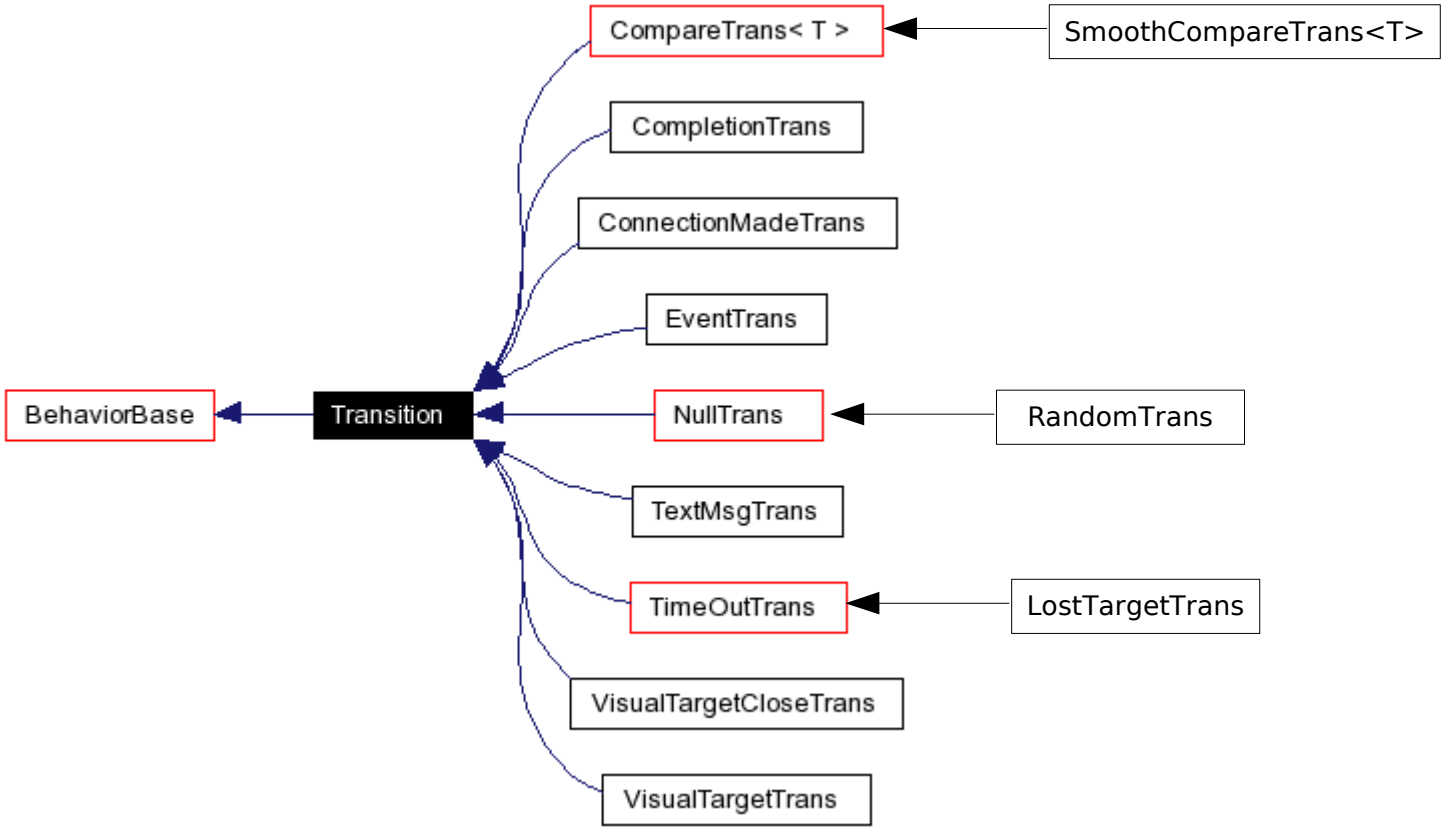
# Types of State Nodes



# 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 note(s).

# Transition Types



# Programs As State Machines

Your program is the parent StateNode:

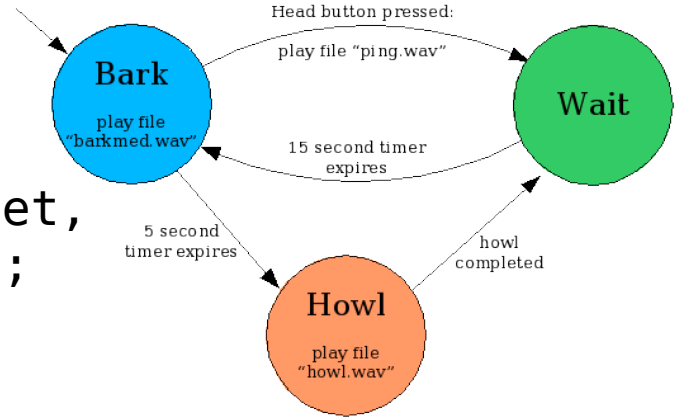
```
#include “Behaviors/StateNode.h”
#include “Behaviors/Nodes/SoundNode.h”
#include “Behaviors/Transitions/CompletionTrans.h”
#include “Behaviors/Transitions/EventTrans.h”
#include “Behaviors/Transitions/TimeOutTrans.h”

class DstBehavior : public StateNode {
protected:
  StateNode *startnode;

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

# 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

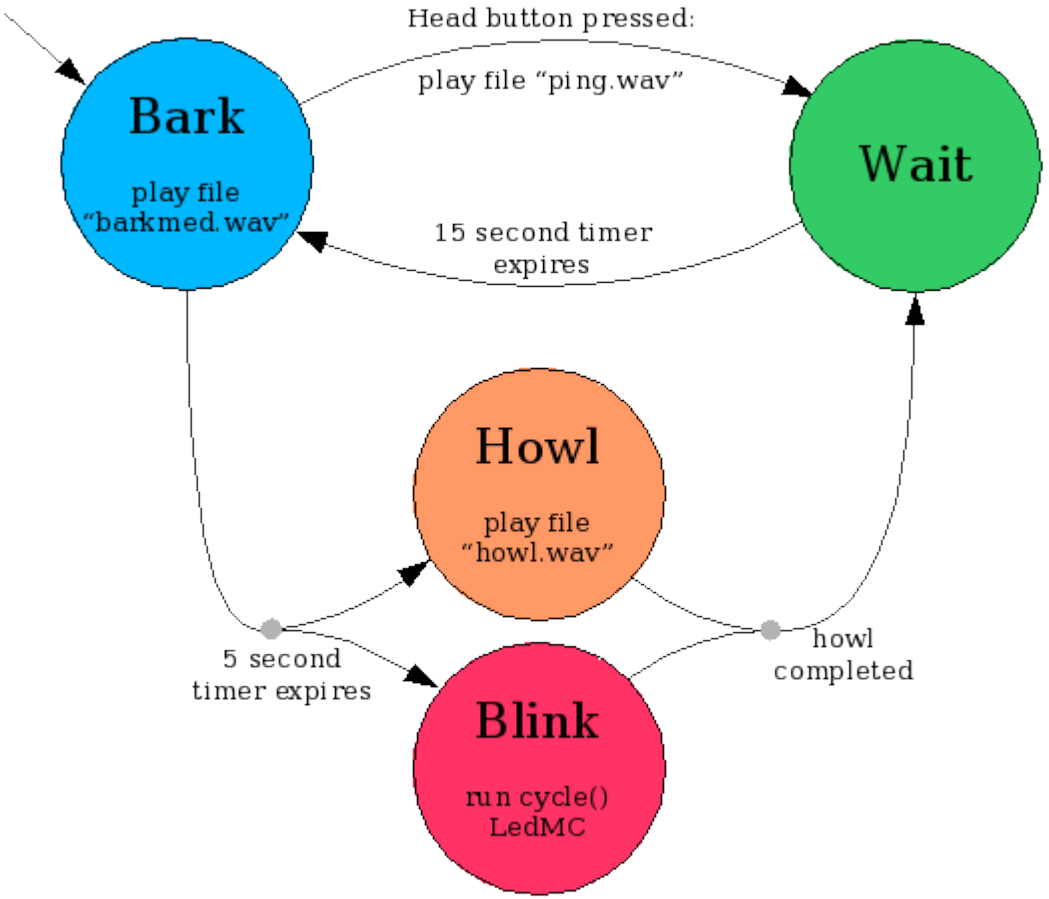
```
  virtual void DoStart() {
    StateNode::DoStart();
    cout << getName() << " is starting up." << endl;
    startnode->DoStart();
  }

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

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.
  - activatedETID 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

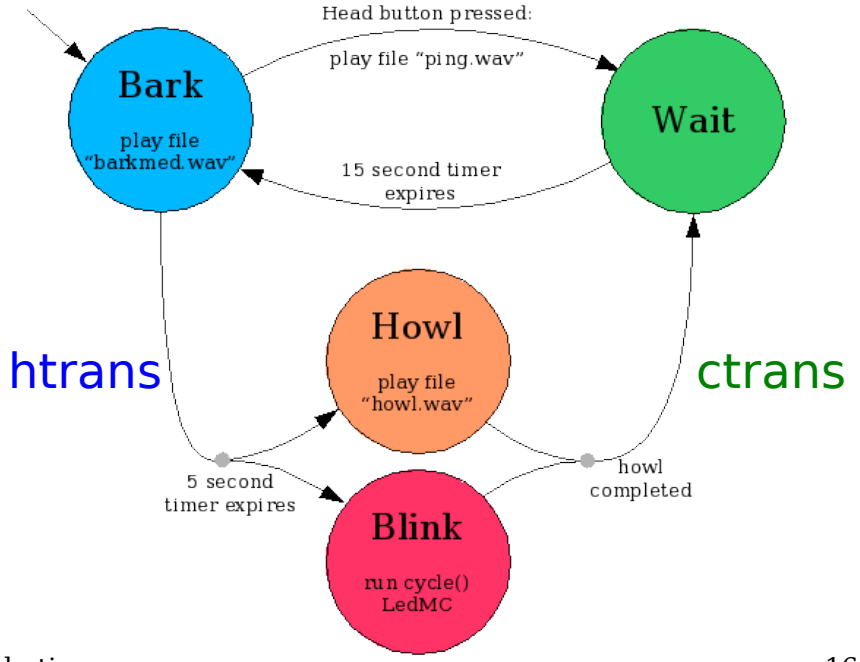


# Blink Using LedEngine::cycle()

- 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

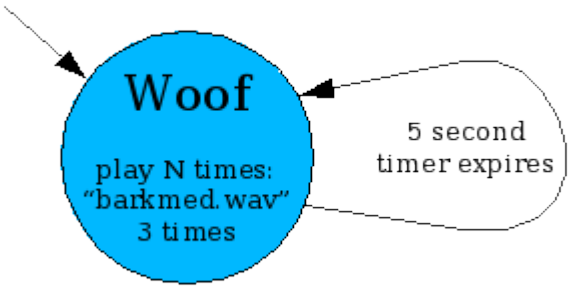
```
private:
  StateNode *startnode;
  SharedObject<LedMC> noblink_mc;
  MotionManager::MC_ID noblink_id;

  virtual void DoStart() {
    StateNode::DoStart();
    std::cout << getName() << " is starting up." << std::endl;
    noblink_mc->set(RobotInfo::FaceLEDMask, 0.0);
    noblink_id = motman->
      addPersistentMotion(noblink_mc,
                          MotionManager::kBackgroundPriority);
    startnode->DoStart();
  }


  virtual void DoStop() {
    std::cout << getName() << " is shutting down." << std::endl;
    motman->removeMotion(noblink_id);
    StateNode::DoStop();
  }
```

The LedNode's motion command has kStdPriority. When the node is deactivated it calls removeMotion. This background motion command then clears the face LEDs.

# Creating New State Node Types

- Only a few types of statenode 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 {
private:
  StateNode* startnode;
public:
  DstBehavior() : StateNode("DstBehavior"), startnode(NULL) {}

  virtual void setup() {
    StateNode::setup();
    sndman->LoadFile("barkmed.wav");
    StateNode* woof_node =
       new PlayNTimesNode("woof","barkmed.wav",3);
    addNode(woof_node);
    woof_node->addTransition(new TimeOutTrans(woof_node,5000));
    startnode = woof_node;
  }

  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
```

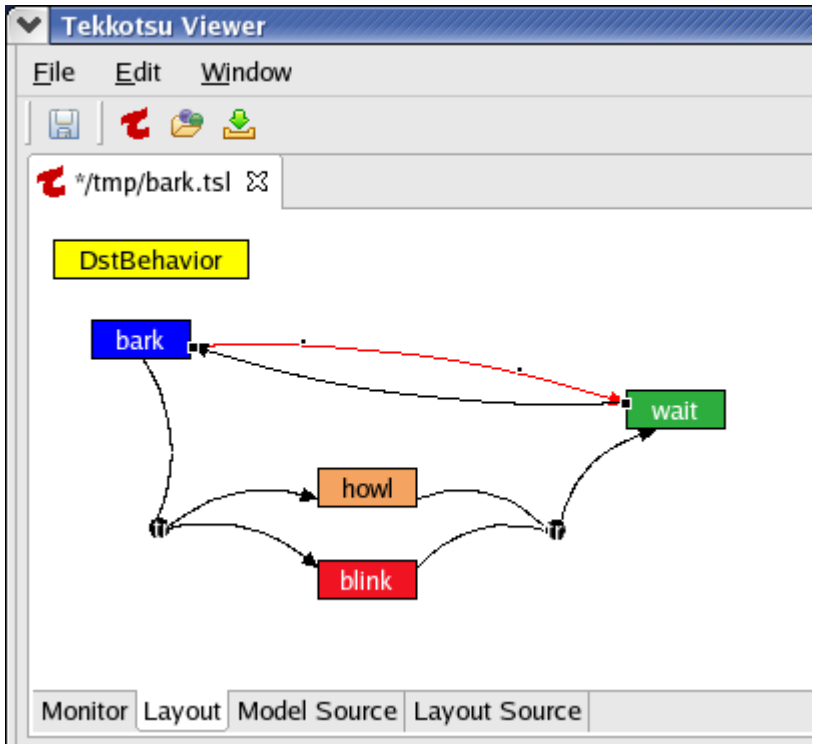
# LostTargetTrans.h

```
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;
}
```

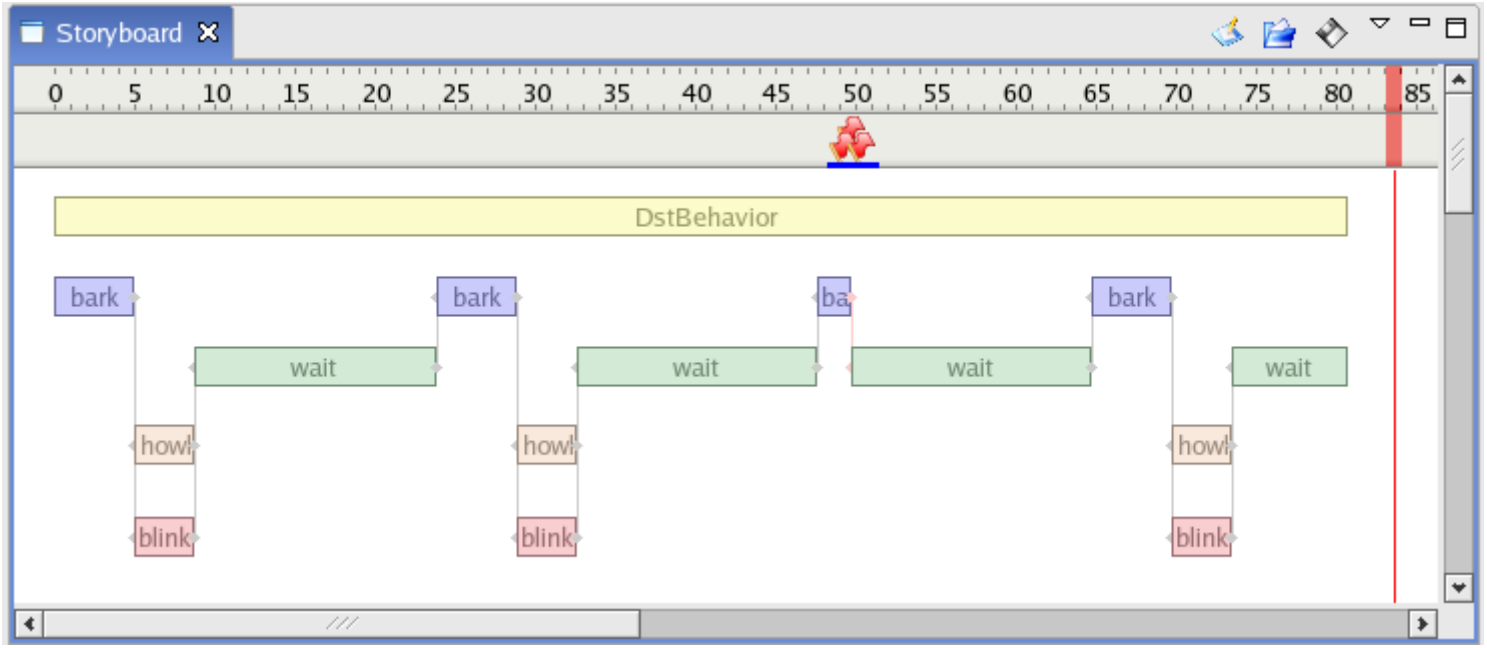
# Other Transition Types

- 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:

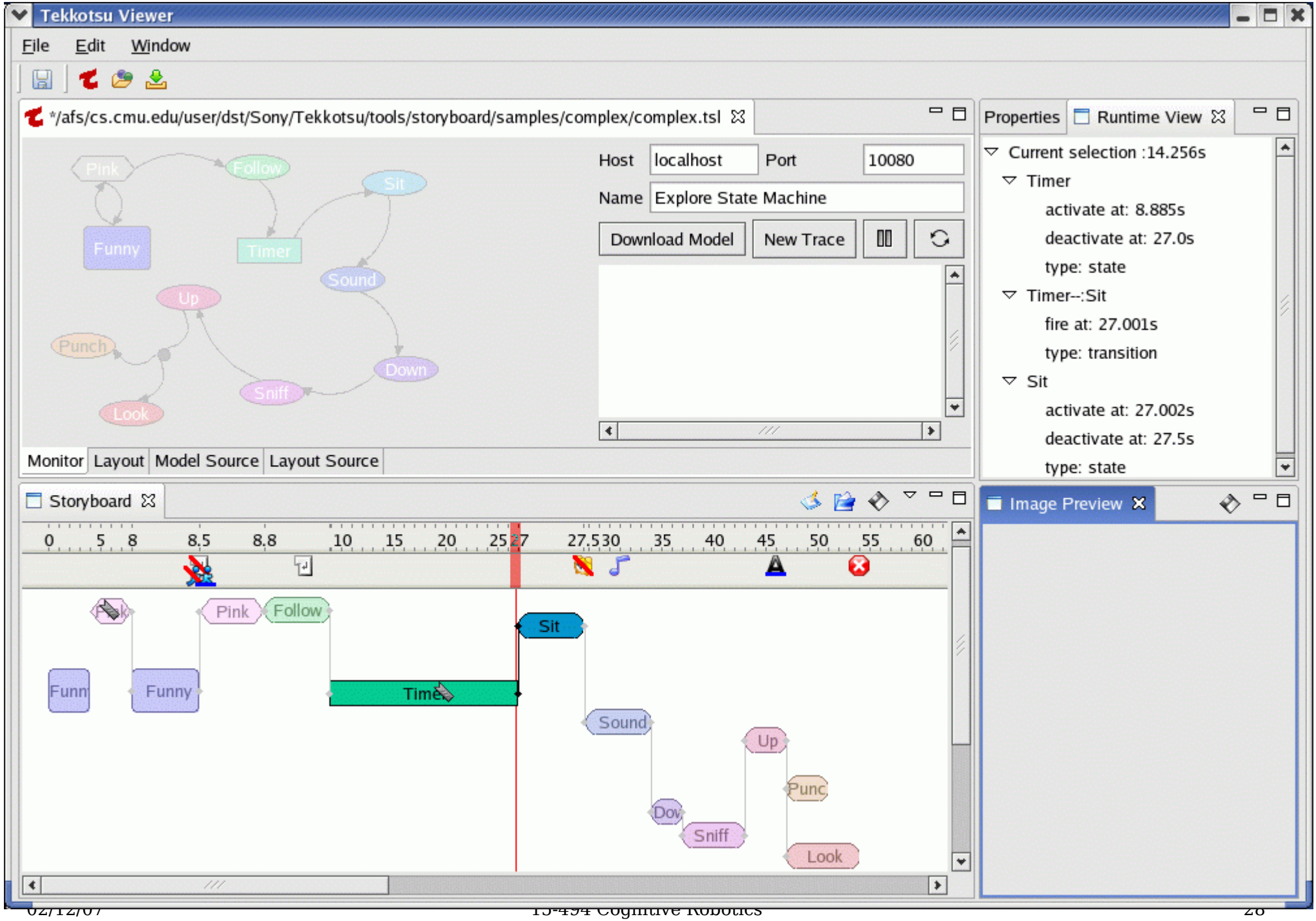
```
CompareTrans(node37,
             &state->sensors[NearIRDistOffset],
             CompareTrans::LT,
             200,
             EventBase(EventBase::sensorEGID,
                       SensorSourceID::UpdatedSID,
                       EventBase::statusETID))
```




# Storyboard Tool: State Machine Layout



# Storyboard Tool: Storyboard Display



# Storyboard Tool: Snapshots