Writing Finite State Automata in Tekkotsu
Shawn Turner – Robotics Seminar - University at Albany

The purpose of this document is to overview and familiarize the reader with how automata can easily be handled using the tools provided with the Tekkotsu framework. A brief technical overview of some key components will be followed by a walk-through of the demo behavior included that best demonstrates how these components can be applied. A minor extension of this demo is included as an example of how a programmer can begin to customize the algorithm for our demonstration purposes.

* Before really attempting to create any automatas using the Tekkotsu framework, it's best if you have completely worked through the tutorials at www.tekkotsu.org. These include Behavior basics, MotionCommand fundamentals, use of the TekkotsuMon tools/GUI, and a little on how Tekkotsu approaches event driven programming.

As a matter of straightening out vocabulary, consider the following simple "Automata"

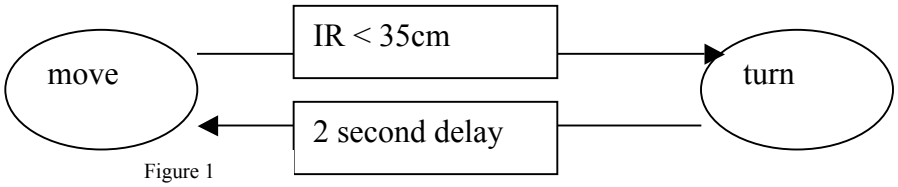


Figure 1

For the purposes of Tekkotsu, the ovals represent "Nodes" and the labeled arrows represent a "Transition", where the direction of the arrow represents the flow of program control. Automata written using Tekkotsu is constructed from classes inherited from such nodes and transitions.

The basic node from which we should build is called *StateNode*.
A StateNode (or just node) is defined as either a "state machine controller as well as a node within a state machine itself". That means depending on how you initialize and construct your network, the individual instance or derivation of your node could serve several different functions. (A more in depth look at state machine theory might be good to clarify that definition) What that means in a practical sense, is that no matter what and how many and what arrangment of nodes we need, they all wind up inheriting from StateNode. StateNode is constructed to allow this flexibility as follows:

```
StateNode:BehaviorBase
public:
...
        DoStart()
        DoStop()
        addTransition(Transition *)
        addNode(StateNode *)
        setup()
…
protected:
        std::vector<StateNode *> nodes;              //nodes in our machine
        std::vector<Transition *> transitions;       //"registered" transitions
```

These pieces make up the bulk of the work that a node is going to do. The "nodes" vector represents nodes that we've "added" to our machine (we're acting as a State Machine Controller in this context). The transitions vector holds all transitions that we've added to his node in our state machine (the other context from above). If you're unfamiliar with the vector class from the Standard Template Library, it's most easily thought of a a linked list. It's important to point out that StateNode is itself, a behavior. So we're obligated to write a DoStart() and DoStop() method just as we would with any other method. It also means that we can trap events here if we choose. The examples below will show why trying to handle events inside the node is not always the preffered way to handle this in tekkotsu. The addTransition and addNode methods above handle the maintenance of the two vectors- these are written in full in StateNode. There's no need to reimplement them in your programs, we'll just call them as we need to. The last function above, setup() , is probably the most critical. This is where allocation and definition of your node network is peformed, including setting MotionCommand ids for your nodes.

If you survived that, here's a concrete look at the demo behavior "ExploreMachine". This behavior moves the Aibo straight until a wall is detected within a certain proximity by the IR sensor, the behavior then randomly decides a direction to turn and begins walking straight again. A graphical look is fairly close to "figure 1 " above.

take a look at:
/usr/local/Tekkotsu/Behaviors/demos/ExploreMachine.h

the essentials:

```
ExploreMachine:StateNode
        ...
```

```
        DoStart()
        DoStop()
        setup()
        ...
protected:
        StateNode * start                   //a reference point
        WalkNode * turn, *move              //for turning, moving
        MotionManager::MC_ID walkid         //a common id for all nodes in our network
```

The full class contains all the usual fair, a couple of constructors, DoStart, DoStop, a couple of housekeeping functions and the critical setup() function.
The only three that really need change from node to node are the DoStart, DoStop, and setup. Let's walk through how these three functions and four variables can cause the behavior that this claims.

This is still a behavior, so after the constructor, DoStart() is the first block that runs:

```
        virtual void DoStart() {
                StateNode::DoStart();
                start->DoStart();
        erouter->addListener(this,EventBase::stateMachineEGID,(unsigned int)turn);
        }
```

The first instruction calls the DoStart method in the base class, this checks to see if *this* is currently active, if it's not, it runs setup() and sets the appropriate housekeeping flags. Following setup() for *this* class, it runs it for the protected StateNode * start, and then registers itself as a listener for the event used for state changes, stateMachineEGID. Looking into this setup function will hopefully show how this fits together:
Explanation is in-code as comments

```
        virtual void setup() {
                //cout << "Explore SETUP " << issetup << "...";

                //this section resolve the RobotModel to it's proper sensor array offset
                //for getting the IR reading
                unsigned int IRDistOffset;
                if(state->robotDesign&WorldState::ERS210Mask)
                        IRDistOffset=ERS210Info::IRDistOffset;
                else if(state->robotDesign&WorldState::ERS220Mask)
                        IRDistOffset=ERS220Info::IRDistOffset;
                else if(state->robotDesign&WorldState::ERS7Mask)
                        IRDistOffset=ERS7Info::NearIRDistOffset;
                else {
                        serr->printf("ExploreMachine: Unsupported model!\n");
                        return;
                }
                //setup and start some motion
```

```
//WalkMC is pre-defined in Tekkotsu/Motion/WalkMC.cc (.h)
//if this looks really unfamiliar, check out the
//"Writing Your First Motion Command" tutorial

SharedObject<WalkMC> walk;
walkid=motman->addMotion(walk);  //walkid is a protected variable


//begin our network setup
//walk, start, and turn are all nodes
//start points to the starting state
//move and turn are of type WalkNode, another derivation of StateNode
// that allows for good interaction with WalkMC

WalkNode * move=NULL;              //initialize
```

/* In this example, the class we're creating (ExploreMachine) is the controller for our Automata. We need to addNodes to it using transitions that will give us the algorithm that we want. The "start" pointer really just holds a reference to where we started and doesn't do much beyond that here.
*/

```
//add and allocate two nodes that we want as part of our network
start=addNode(turn=new WalkNode(0,0,0.5f,this));
addNode(move=new WalkNode(150,0,0,this));

//set the names for consistency's sake
turn->setName(getName()+"::turn");
move->setName(getName()+"::move");
```

/* So at this point we've got two nodes, turn and move. Their particular attributes are defined in the class from which they were created, in this case, WalkNode. But we still haven't defined any sort of direction or logic to how they should interact.
Enter the Transition. Transitions are defined (by tekkotsu) as being "smart" (they handle their own activation / deactivation). There's more to be said about Transitions and how they work, take a look at Tekkotsu\Behaviors\Transition.h for more info. There are two basic properties that we need concern ourself with. A Transition contains a "destination (s)" and some sort of triggering condition that acts as an event
*/

```
//this sample is very simple, so each node needs only one transition

    turn->addTransition(new TimeOutTrans(move,2000));
```

/* This says that in the turn node, "register" a Transition, in this case a TimeOutTrans. The first argument "move" represents the target node to which control should be transferred, the second a timeout setting in milliseconds.

Take another look at the figure 1 at the top of this document, you'll see this transition conforms to that definition.
*/

```
//this is all one line
move->addTransition(new SmoothCompareTrans<float>(turn,&state-
>sensors[IRDistOffset],CompareTrans<float>::LT,350,EventBase(EventBase::sensorEGI
D,SensorSourceID::UpdatedSID,EventBase::statusETID),.7));

/* Similarly to the turn node, set a target and "trigger" here. This call is a little more
messy, but it boils down to setting control to the "turn" node and defining a certain IR
minimum as the triggering event. There are plenty other transitions as well that are worth
looking at.
*/
                /* setWalkID basically sets a MotionManager MCID inside the WalkNode
class for Motion handling. Note how both nodes get the same ID, walkid. This means that
both nodes are talking to the same running motion command, passing it updated
arguments to keep it moving.
*/

                turn->setWalkID(walkid);
                move->setWalkID(walkid);
                StateNode::setup();             //important! sets issetup flag to true.

                //cout << issetup << endl;
        }       //end setup
```

So that's it? Not quite. Given the simplicity of the problem and the well defined WalkNode, thats the meat of what you need to worry about. But a question that you should maybe be asking is, how do the actuators know how and when to update in order to be walking in one state and turning into another. The answer is in how WalkNode is designed. This could just as easily be any node that you may need that handles changing motions between another node.

* An important note about Transitions, when a transition activate() 's itself, it shuts down (runs DoStop) on the node that is currently running. That means two things- One, DoStart() will certainly get called again when control transfers back, and Two, you won't be able to have an persistent data as part of the nodes in your network. Data structures, static variables, event registeration, and the like will be reset the next time you return to the node. The easiest way that seems to be handled is by putting any such components in the node acting as the State Controller. In the case of our example above, ExploreMachine.

Recall that WalkNode derives from StateNode, and StateNode derives from BehaviorBase. So every time a WalkNode is started (when a state change is triggered), DoStart() will need to run. WalkNode works with that in mind:

```
virtual void DoStart() {
	StateNode::DoStart();
	updateWalk(x,y,a);
}
```

The x, y, and a arguments are supplied by the constructor or various other member functions- These specify how the WalkMC is to behave, x represents forward, y represents lateral movement, and a specifies kind of an "arc" angle. I'd welcome a clearer interperation of how that motion processes those arguments. Another quick glance at the ExploreMachine setup() above, and you can see how the constructor is used to make one node hold a "straight" walking vector, and one an attempts at a "rotating" one.

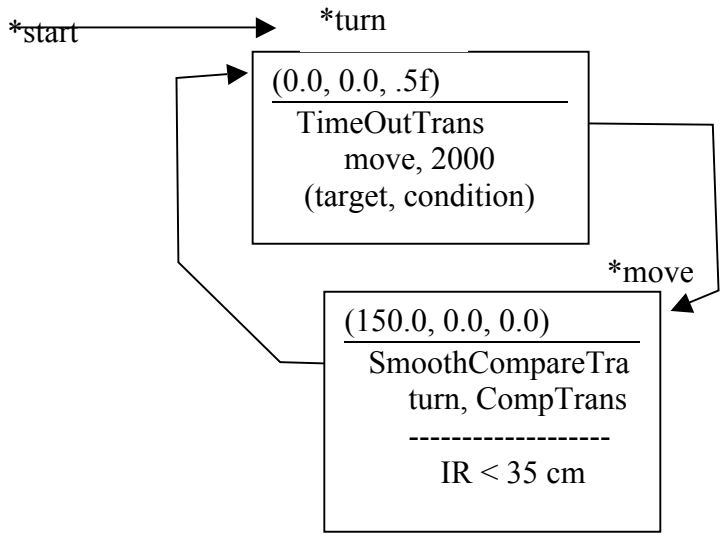
updateWalk(x,y,a) is protected and actually makes the call to motman or a MMAccessor to get things to move:

```
//!if the walk is invalid, create; then set xya
//invalid_MC_ID is expected and set by our class often
void updateWalk(float xvel, float yvel, float avel) {
	if(walkid==MotionManager::invalid_MC_ID) {
		SharedObject<WalkMC> walk;
		walk->setTargetVelocity(xvel,yvel,avel);
		walkid=motman->addMotion(walk);
		walkidIsMine=true;
	} else {
		MMAccessor<WalkMC> walk(walkid);
		walk->setTargetVelocity(xvel,yvel,avel);
	}
}
```

This much should be pretty straightforward from dealing with MotionCommands. setTargetVelocity does exactly what you think it would, sets values for the "vector" used in moving the Aibo. There's a great deal more to WalkNode, take a look at:

Tekkotsu/Behaviors/Nodes/WalkNode.h

Here's another look at how this same network would function, knowing what we do know about the node / transition relationship.



That wraps up the overview of their demo, now let's extend it to include one more node that can get the aibo to walk straight down say a hallway, and check each side at regular intervals to help detect what might be a door.

part II to come.