Introduction to Behaviors using Tekkotsu

Shawn Turner – Robotics Seminar - University at Albany – Spring 2004

The purpose of this document is to give an overview of an Aibo behavior in general, and more specifically, how to implement such a behavior using the Tekkotsu API. The is no assumption of prior experience programming with Tekkotsu/OPENR, but the reader is expected to be familiar with OOP using C++.

The document is broken down into four major partitions. First, an explanation of a behavior and how it interacts at a high level with the run-time Aibo. Second, a technical breakdown of the data members and methods that you'll need to be concerned with in order to start customizing your own behaviors. Third, a simple example will be walked through to hopefully solidy what needs to be done. And fourth, instructions on how to add a new behavior into the Tekkotsu "project".

What is a behavior?

One way to define behavior would be to say that a behavior is how a robot creates action from perception as shown in the diagram below:
(http://www-2.cs.cmu.edu/~robosoccer/cmrobobits/lectures/Introduction.ppt)

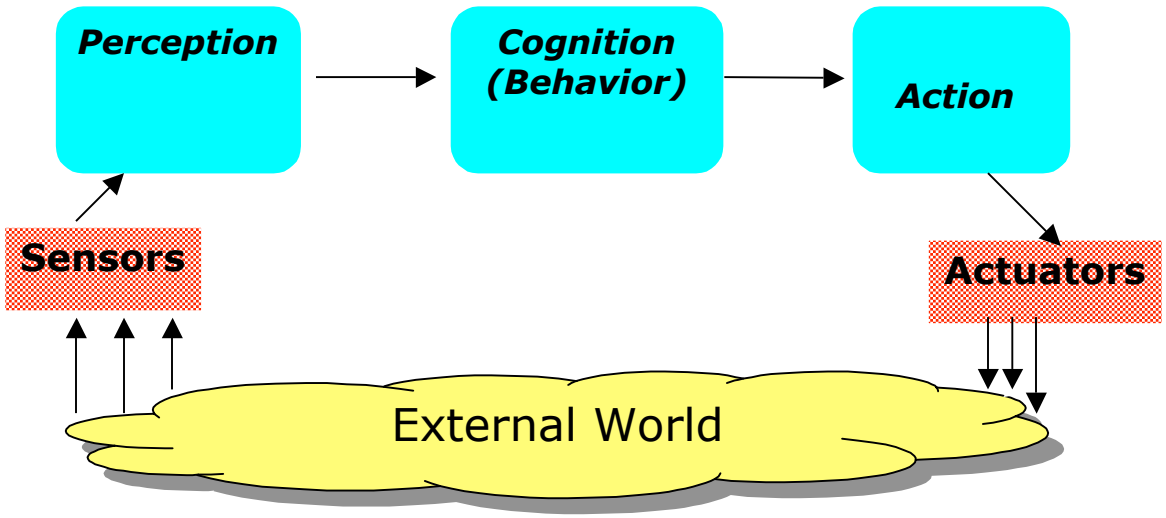


Figure 1

The idea being, that the sensors on the robot detect the "world" from the robot's point of view, a perception module abstracts that into something that a programmer could find meaningful, then the Behavior portion uses that data to make some decisions, and based on those decisions, an "action" module tells the actuators (motors) on the robot to move with some amount of torque.

The Tekkotsu API simplifies this process greatly by giving the programmer tools which can easily catch the "perception" and take the "action". In other words, software writing in this framework is completely abstracted from the hardware. For instance, if we wanted to get the current IR (infared) distance reading, we need only include the proper header files and this line of code:

```
float distance = state->sensors[IRDistOffset];
```

This is a great deal easier then if we needed to worry about things like the actual hardware offset for that sensor and handling the sensor frame-rate (how fast we're actally

receiving data from the hardware at a given time). All we worry about is that we want a sensor, so we access the *sensors* array appropriately.

Similarly, the "action" portion of the algorithm, is equally abstracted in the form of a MotionCommand. There's no need to get into the details of a MotionCommand here (there's a little in the example), just know that a MotionCommand is the portion of the code that actually makes the actuators move. It's the job of the Behavior to initialize MotionCommands, and to turn them on and off.

Hopefully, that makes the role of the behavior clear. We're concerned only with the logical input and output of data, not in how it's transferred to the physical world. In this context, Tekkotsu behaviors are analagous to a "main" method in C or C++.

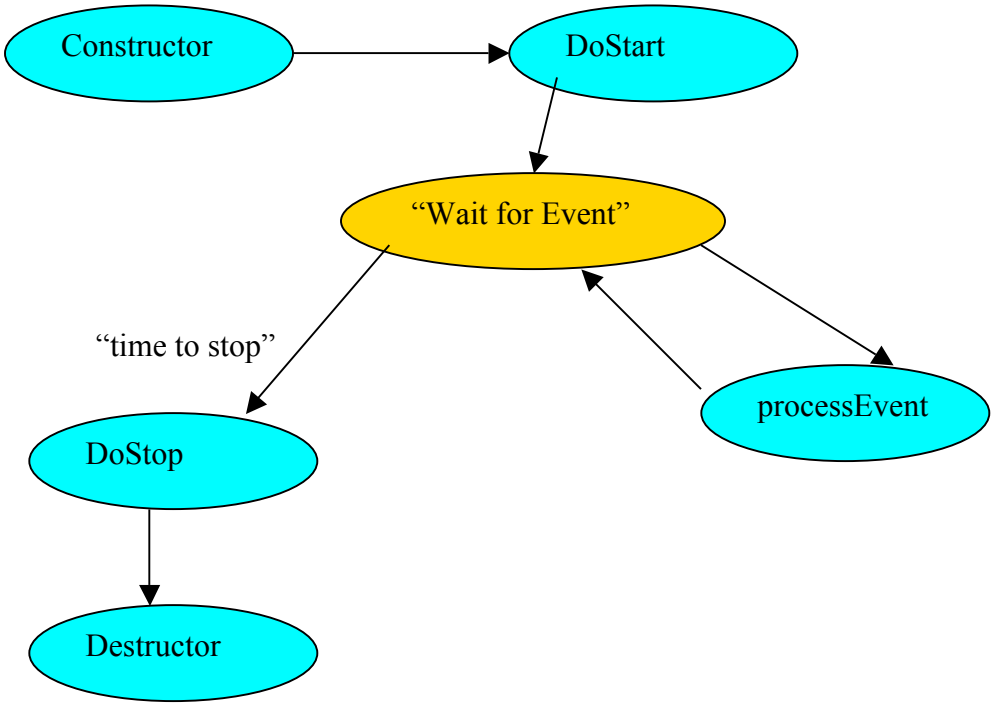
How do we build it?

Unlike a "main" method in C++, which runs sequentially from top to bottom until some sentinal condition is met, Behaviors are *event driven*. Segments of code in event-driven programming are not always executed from top to bottom, segments are designated to start when a certain condition is seen by the program. These conditions on which action driven programs rely are commonly referred to as *events*. In context to "figure 1" above, events would mostly fall within the "perception" category of the Aibo's execution. So you can think of a program running on an Aibo as sitting in some infinite loop, waiting for the system to turn some external perception into an *event.*

But where does execution start? Before that gets answered, let's lay out the necessary functions that need defining for a Behavior to be valid.

```
class TutorialBehavior: public BehaviorBase{
        TutorialBehavior();             //the constructor
        ~ TutorialBehavior();           //the desctructor
        void DoStart();
        void DoStop();
        void processEvent(const EventBase &)
        std::string getName() const;
};              //end behavior
```

This *is* still C++ OOP, so the first thing that would get executed would be the constructor, likewise, the destructor is still the last block to get executed.
There is a "life-cycle" to Tekkotsu behaviors:



Behavior Life-Cycle

The "Wait for Event" stage is not a real module we need to write, simply represents the higher-level event router which sends us the messages. The other five modules however, are something that we need to code. If you're at all familiar with Java applet programming, the architecture is identical.

Are we there yet?

Pretty much. If you've followed the outline up to here, we know when and where our program starts, when it executes what code, and what code runs when the system terminates our behavior. So let's put some code into this framework.
What follows is an extensively commented version of running behavior code (there's a less commented version at the back of this doc). Please note, there's one more necessary change that you need to make to your Tekkotsu "project" before you can compile and run this behavior. Explanation of how to setup and create a project follows this code sample. All Tekkotsu behaviors should inherit from *BehaviorBase.*

```
//-*-c++-*-
//Shawn Turner
//Aibo will walk straight until a barrier is detected by the IR sensor
//Aibo will then remain stopped until the barrier is no longer detected
//Good simple behavior example, CMU's RoboBits has a similar OPENR newbie
//assignment using the g-sensor (gravity sensor)


#ifndef INCLUDED_TutorialBehavior_h_
#define INCLUDED_TutorialBehavior_h_

#include "Behaviors/BehaviorBase.h"                    //necessary for all behavior
#include "SoundPlay/SoundManager.h"                    //necessary for any sound
#include "Motion/MotionManager.h"                      //for motions
#include "Motion/PostureMC.h"                          //unneccesary for this example
#include "Motion/WalkMC.h"                             //definition of the basic walk
#include "Shared/WorldState.h"                         //for “state” pointer
#include "Motion/MMAccessor.h"                         //to access active motions
#include "Shared/SharedObject.h"                       //always with motion
#include "Events/EventRouter.h"                        //for event handling (always)

#define IROORDist 900                                  //maximum 210 range (mm)
#define IRMinDist 300                                  //let's call this the minimum (mm)

class TutorialBehavior : public BehaviorBase {
        public:

        /*
         The constructor / destructor should be old news to you. You should implement
        housekeeping up and down here.
        */

        TutorialBehavior():BehaviorBase()
         {
                  //do nothing
         }
        ~TutorialBehavior()
        {
                //if somehow we’re still running when this runs, we should be sure to turn it off
                if(isActive())
                  DoStop();
        }


        //  DoStart is typically where the setup logic / motion initializations occur

        virtual void DoStart() {
                //call superclass first for housekeeping:
                BehaviorBase::DoStart();




        /*
```

```
		setup and start a motion command.
		Details on this can be found in the "Writing your first MotionCommand" tutorial
		on Tekkotsu.org . the addMotion method is the equivalent of "turning on" the motoin. In
		this case, the WalkMC.
		*/

		walk_id=motman->addMotion(SharedObject<WalkMC>());

		/*
		any event that this behavior cares about needs to be "registered" with the event router.
		This tells the event router to send us a message if it detects such an event.
		In this case, I want to know if any sensor data has changed.

		A good list of pre-defined  trappable events can be found here:
		http://www-2.cs.cmu.edu/~tekkotsu/dox/classEventBase.html

		*/
		erouter->addListener(this,EventBase::sensorEGID); // subscribes to all sensor events

} //end DoStart

/*
		DoStop basically undoes all the setup found in DoStart
		Should be fairly straightforward
*/
virtual void DoStop() {
		motman->removeMotion(walk_id);		//remove the WalkMC motion tied to walk_id
		erouter->forgetListener(this); // stops getting events (and timers, if we had any)
		BehaviorBase::DoStop();	// run base class for consistency / cleanup
}

/*
 processEvent really does all the work here. We want Aibo to walk as so long as there is nothing
directly in it's path. I've used the IR sensor set at an arbitrary minimum to detect an obstacle and
effectively stop walking

processEvent runs anytime an event we registered for is detected. Because we can register for
as many events as we like, it's a good idea to use a switch or if statement to detect exactly which
event we're handling, as well as if we somehow wound up with an event that we didn't think we
registered for (debugging)
*/

virtual void processEvent(const EventBase& event) {
		float distance;

		//get the current distance from the IR sensor
		distance = state->sensors[IRDistOffset];

		//cout<<"distance: "<<distance<<endl;

/*
the MMAccessor let's us access a currently running MotionCommand of a certain type by it's
MotionManager::MC_ID. We defined our MC_ID inside of DoStart. Again ,look at the
MotionCommand tutorial for a little more detail.

It's worth noting that MMAccessor is a template expecting a MotionCommand. Make sure that
you're accessor definition matches whatever motion command you're actually trying to effect and
that's tied to your MC_ID (walk_id here).
```

```
*/
        MMAccessor<WalkMC> walk(walk_id);

        //switch to guarantee we're handling the correct event

        if(event.getGeneratorID() == EventBase::sensorEGID){
                if(distance <= IRMinDist)
                        //update the walk, this effectively stop Aibo
                        walk.mc()->setTargetVelocity(0.0, 0, 0);
                else
                        //update the walk on a "dead ahead" vector
                        walk.mc()->setTargetVelocity(150.0, 0, 0);
        }
        else
        //this should never happen, but can be real helpful when we forget to register correctly!
                cout << "Bad Event:" << event.getName() << endl;

  }

//return an approriate string representing the name. This too, is neccessary.
virtual std::string getName() const { return "TutorialBehavior"; }

protected:
         MotionManager::MC_ID walk_id;          //create an MC_ID for my walk


};                  //end class

#endif
```

That's it! The behavior is finished. Now to set it up in a project.

Project Implementation:

What follows are just instructions on how to add this behavior into a Tekkotsu project. Assuming that your userid is "stewie" and the installation paths have been setup in the default way.

you should already have a /home/stewie home directory

```
%> cd /usr/local/Tekkotsu
%> cp –R project /home/stewie
```

this makes a private copy of projet specific files into your home directory

```
%> cd /home/stewie/project
```

take a look at the current contents of the directory, you'll see a whole bunch of setup files. Copy the TutorialBehavior.h file into your new project directory first.

the file that we're primarily concerned with is called "StartupBehavior_SetupModeSwitch.cc", open it in your favorite editor

%> emacs StartupBehavior_SetupModeSwitch.cc

there's a long list on includes at the top of the file, add the new behavior at the bottom of the other includes:

#include "TutorialBehavior.h"

The Tekkotsu project works as a menu system where multiple behaviors can all be loaded simultaneously, but the user can get control of which behaviors are on at any given time. This is great for debugging / comparison. We need to add our new behavior into the menu system accordingly:

StartupBehavior_SetupModeSwitch.cc:

```
...//some header files
//added by stewie
#include "TutorialBehavior.h"

ControlBase*
StartupBehavior::SetupModeSwitch() {
addItem(new ControlBase("Mode Switch","Contains the \"major\" applications - mutually exclusive
selection"));
	startSubMenu();
	{
	//this group allows the behaviors to turn each other off when you start a new one so
	//only one is running at a time - like radio buttons
	BehaviorSwitchControlBase::BehaviorGroup * bg = new
	BehaviorSwitchControlBase::BehaviorGroup();

	//put behaviors here:
	if(state->robotDesign&WorldState::ERS210Mask) //this one only really works on the 210
	addItem(new BehaviorSwitchControl<AlanBehavior>("AlanBehavior",bg,false));

addItem(new BehaviorSwitchControl<FollowHeadBehavior>("FollowHeadBehavior",bg,false));
addItem(new BehaviorSwitchControl<SoundTestBehavior>("SoundTestBehavior",bg,false));
addItem(new BehaviorSwitchControl<ChaseBallBehavior>("ChaseBallBehavior",bg,false));

/*
 this actually adds our new behavior into the menu system
 the bg argument represents the current "behavior group" that we're associating with, the last
argument of false, is a flag whether or not to keep the object in memory once it's been shut off by
the menu.
*/
//added by stewie
addItem(new BehaviorSwitchControl<TutorialBehavior>("TutorialBehavior",bg,false));

...//there's a bunch more here
	}
	return endSubMenu();
}
```

At this point, we've created a valid behavior and added it into the Tekkotsu framework, all that's left is to compile.

from your project directory:

/home/stewie/project %> make install

Be sure that the memory stick is correctly mounted to /mnt/memstick before you try and compile.

umount the memory stick, stick it in the Aibo, and we're ready to go!

Instructions on how to access / use the Tekkotsu menu monitoring system (a neccessity if you want to turn anything useful on) can be found here:

http://www-2.cs.cmu.edu/~tekkotsu/TekkotsuMon.html

Questions and Suggestions should be sent to st2750@albany.edu

```
//-*-c++-*-
//Shawn Turner
//Aibo will walk straight until a barrier is detected by the IR sensor
//Aibo will then remain stopped until the barrier is no longer detected
//Good simple behavior example, CMU's RoboBits has a similar OPENR newbie
//assignment using the g-sensor (gravity sensor)


#ifndef INCLUDED_TutorialBehavior_h_
#define INCLUDED_TutorialBehavior_h_

#include "Behaviors/BehaviorBase.h"                     //necessary for all behavior
#include "SoundPlay/SoundManager.h"                     //necessary for any sound
#include "Motion/MotionManager.h"                       //for motions
#include "Motion/PostureMC.h"                           //unneccesary for this example
#include "Motion/WalkMC.h"                                      //definition of the basic walk
#include "Shared/WorldState.h"                          //for state pointer
#include "Motion/MMAccessor.h"                                  //to access active motions
#include "Shared/SharedObject.h"                        //always
#include "Events/EventRouter.h"                                 //for event handling

#define IROORDist 900                                           //maximum 210 range
#define IRMinDist 300                                           //let's call this the minimum

class TutorialBehavior : public BehaviorBase {
        public:
        TutorialBehavior():BehaviorBase()
         {
                  //do nothing
         }
        ~TutorialBehavior()
        {
                        if(isActive())
                                DoStop();
        }
        virtual void DoStart() {
                //call superclass first for housekeeping:
          BehaviorBase::DoStart();
          walk_id=motman->addMotion(SharedObject<WalkMC>());
          erouter->addListener(this,EventBase::sensorEGID); // subscribes to all sensor events
        }

        virtual void DoStop() {
                motman->removeMotion(walk_id);
                erouter->forgetListener(this); // stops getting events (and timers, if we had any)
                BehaviorBase::DoStop();
                }


        virtual void processEvent(const EventBase& event) {
                float distance;
                unsigned int capture=RFrLegOffset;
                distance = state->sensors[IRDistOffset];
                //cout<<"distance: "<<distance<<endl;

                MMAccessor<WalkMC> walk(walk_id);
                if(event.getGeneratorID() == EventBase::sensorEGID){
                        if(distance <= IRMinDist)
```

```
                        walk.mc()->setTargetVelocity(0.0, 0, 0);
                else
                        walk.mc()->setTargetVelocity(150.0, 0, 0);
            }
            else
                cout << "Bad Event:" << event.getName() << endl;


          } //end processEvent

        virtual std::string getName() const { return "TutorialBehavior"; }

        protected:
         MotionManager::MC_ID walk_id;


};              //end class

#endif
```