Tutorial 2
Simple Agent Action Tutorial: Speaking

In this tutorial, we'll look at creating and performing an action. As the PAR system stands, only agents can perform actions. This action is one that the agent will perform by itself (and not require the use of another object).

Creating an Action

The PAR system requires that an action have preconditions and termination conditions defined. Applicability conditions, such as testing the agent's capabilities and the execution steps are also needed. For this example, we're going to have the agent speak. First, we need to create four python scripts that tell the par system everything it needs to know about the action. (These files can be found in the PAR/actions directory). We also include an example (Speak.py) that shows how to combine all actions into one file. Using either one or four files will not effect the execution of the code.

Create Python Scripts:

SpeakApp.py (Applicability Script)
SpeakPre.py (Prerequisites Script)
SpeakExec.py (Execution Script)
SpeakCul.py (Termination Script)

The first script we create, SpeakApp.py tells the system what qualifies an agent to perform an action. In this case, we need to make sure the parameter we give is an agent (and not an object) and that the agent has this action (or a parent action) as a capability within the database. Since the agent doesn't require any other object to perform this action, only the agent needs to be sent as an argument. For this script, all we need is:

```
def applicability_condition(self, agent):
        if checkCapability(agent, self.id):
                return SUCCESS
        else:
                return FALURE
```

For this tutorial, we're going to assume that there is nothing the agent has to do before speaking. Therefore, our prerequisites script will contain the following to indicate that the action is ready to be performed:

```
def preparatory_spec(self, agent):
        return SUCCESS
```

Next, we need to say what steps are required for the action to execute. Since this action is a base action (it does not rely on any other action), we create a python dictionary that labels this action a PRIMITIVE action.

The python script for SpeakExec will then look like:

```
def execution_steps(self, agent):
        setProperty(agent,"obj_status","OPERATING");
        actions = {'PRIMITIVE':("Speak",{'agents':agent})};
        return actions
```

Here, we create a tuple explaining to the system everything we need to know to perform this action. We have an action called speak that involves only an agent. It's assumed that the connecting piece of software, such as a rendering engine, has some understanding of how to perform or animate Speak. Finally, since the termination of the action occurs when we are finished speaking, we can simply use the allotted time to decide when the action is finished. Within the SpeakCul.py script, write:

```
def culmination_condition(self, agent):
        if self.start_time+self.duration > getElapsedTime():
                setProperty(agent,"obj\_status","IDLE");
                return SUCCESS
        else:
                return INCOMPLETE
```

Now that we have python scripts that explain the logic of the speaking action, we can add this action to the database. Execute this line within your mysql database:

```
INSERT INTO action ( act_name, act_appl_cond, act_term_cond, act_prep_spec,
        act_exec_steps, act_obj_num, act_parent_act_id)
        values (
                'Speak',
                'SpeakApp.py',
                'SpeakCul.py',
                'SpeakPre.py',
                'SpeakExec.py',
                -1,
                8);
```

If all action definitions are kept in the same file, the mysql insert statement will appear as

```
INSERT INTO action ( act_name, act_appl_cond, act_term_cond, act_prep_spec,
        act_exec_steps, act_obj_num, act_parent_act_id)
        values (
                'Speak',
                'Speak.py',
                'Speak.py',
                'Speak.py',
                'Speak.py',
                -1,
                8);
```

This line adds our action to the action table in the database. act_name is the name of our action. The next four values (act_appl_cond, ac_term_cond, act_prep_spec, act_exec_steps) point to the location of our python scripts. You may need to change the path to correctly point to the scripts. act_obj_num tells the library that we expect one object to interact with this action. Lastly, act_parent_act_id tells the library that this action is a child of another action (in this case, the action is an arm action). Note that, depending on your setup and the action you are creating, the action may not have a parent value (in which case, the value would be -999).

Now that we have an action in the database, we need to connect this action to some real performance code. While this step is not absolutely necessary to test the iPAR system, it does not make sense to run actions within the system that do nothing. For this section, we're going to add a program file called helperFunctions.h. Within this code, add the following functions as headers:

```
#include "agentproc.h"
#include "lwnets.h"
#include "interpy.h"
```

The first header is the same that was used in the basic agent tutorial. lwnet.h, a new header, provides classes and functions that allow us to advance actions through the PaT-Nets. Interpy.h allows us to connect the Python logic code to the C++ PAR representation. Next, we need to add some variables that we'll soon use. These variables are all defined elsewhere in the code, and so, these variables are external dependencies. Add the follow variables to your code:

```
extern Actionary *actionary;
extern ActionTable actionTable; //Holds the mapping of actions to performance code
```

Finally, we can add some functionality to the code. In order to connect actions to real performance code, we require a function to execute some meaningful code when the action is being run. To speak, all we need to the agent to do is print something out to the command prompt (it could issue a call to text-to-speech software and be parameterized to say different things). So, we can write the speak function as:

```
int doSpeak(iPAR *ipar){
        std::string  subject = ipar->getAgent()->getObjectName();
        printf("Hello World.  My name is %s",subject.c_str());
return 1;
}
```

The first line of the doSpeak function retrieves the agent's name performing the command. Also, we have the agent speak by having it print out to the command prompt. Whenever an agent is to perform a Speak action, this function will be called.

Now that we have function to perform our action, we need to set up the actionTable to link the PAR action to this function. To do this, we are going to create a new function called setUpActionTable. Within your code, place the following lines:

```
void setUpActionTable(){
        if (actionary == NULL) {
                actionary=new Actionary();
                actionary->init();
        }
        initprop();
        actionTable.addFunctions("Speak",&doSpeak);
}
```

The first three lines of this function make sure the Actionary is set up, and is identical to how we set up the Actionary in the basic agent tutorial. The next line creates many helpful functions to link Python functions we can use in our action files to C code. The last line utilizes the actionTable, and connects the name of our action in the database to the code it should process when it runs.

EVERY PERFORMABLE ACTION IN THE ACTIONARY NEEDS TO HAVE SUCH AN ENTRY IN THE ACTION TABLE.

Now create a file called main.cpp. Much of this file is identical to the first tutorial. The main function for main.cpp can be rewritten as:

```
1  int main(void){
2          partime = new parTime();    // setup the timing info for the simulation
3          partime->setTimeOffset(8,30,30);    // hours, minutes, seconds from midnight
4          partime->setTimeRate(1);    // how fast should time change
5          setUpActionTable();    // Builds the action table
6
7          //Builds a new agent and gives him HumanAction capabilities
8          agent = new AgentProc("Agent_0");
9          agent->setCapability("HumanAction");
10
11         // Creates an iPAR
12         iPAR* iparTest = new iPAR("Speak", agent->getName());
13         if (iparTest == NULL)
14         {
15                 printf("ERROR: iPAR could not be created.");
16                 return 0;
17         }
18         iparTest->setDuration(10);        // This should run for 10 seconds
19         iparTest->setStartTime(partime->getCurrentTime()); //Set the start time for now
20         iparTest->setPriority(1);
21
22         // Add this iPAR to the agent's action queue for processing and execution
23         agent->addAction(iparTest);
24
25         // Advance the PaTNets so that the action gets popped from the queue,
26         //        processes, and executed.
27         int error = 0;
28         for(int i=0; i<100; i++)
29                 LWNetList::advance(&error);
30
31         printf("\nFinished the simulation\n");
32
33         system("PAUSE");
34 }
```

Much of this function is the same as in the basic agent tutorial. However, it should be noted that setting up the Actionary has been moved into the setUpActionTable. Lines 7-12 show how to build a basic initialized PAR (iPAR), by giving it an agent to perform the action. We also give the iPAR a starting time, a duration (which is used in the termination script), a priority, and a purpose. In this case the purpose field is just being used to clean-up the database. We do not want the database to become filled with old iPARs, so the purpose field is used as a flag to delete iPARs that are not to be performed in the next run of the simulation. Next, we add the iPAR to the agent's action queue and then advance the PaTNets using LWNetList::advance function to pop the action from the queue, process it, and ultimate call for its performance. (Note that for simplicity we've just put the advance call inside a for loop. In most applications, advance should instead be called at every time step of the simulation.

Finally, we want to note that PAR actions are meant to be general, universal definitions. If properly crafted, they are scenario independent. As the Actionary is populated with action (and objects) fewer and fewer should be needed for new scenarios, so the effort required will diminish.