Tutorial 6
Advanced Actions

This tutorial introduces the user to authoring actions that contain pre-requisites, as well as post conditions. It is assumed the user has experience creating basic actions (see the basic action tutorial). For this tutorial, we will be creating a walk action, where an agent in an unknown pose and a known position must traverse to an object with a known position.

Creating the Actions

In the basic tutorial, we created four python scripts (applicability, pre-conditions, execution, and termination) to represent our action. Let's start by creating four scripts for walk. For this tutorial, the scripts names will be:

walkapp.py
walkpre.py
walkexec.py
walkcul.py

For this action, we wish for our agent to traverse to an object in a known location. Therefore, all of the scripts we write should include information about the object, as well as the agent. For Walking, this means our applicability condition script is

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

The only difference between this script and the applicability script in the basic tutorial is that we pass the object in this one.

For our pre-conditions, we want to make sure the agent is standing before we start walking (otherwise, the agent's walk would appear quite strange). In order to successfully have a posture, we need to define postures within the database. Tutorial six explains how to create new property tables. For our example, we can assume that the agent has at least two posture states, standing and not standing. To test properties within the pre-conditions, we use the built in python function getProperty. For all the python functions included with PAR, see Included Python Functions.

```
def preparatory_spec(self, agent,obj):
        posture = getProperty(agent,"obj_posture")
        if(posture != 'STAND'):
                actions = {'PRIMITIVE':("Stand",{'agents':agent})};
                return actions
        else:
                return SUCESS
```

This script does a few things, so let's look at it in depth. Just like in the last script, we passed in both the agent and the object. Next, we use the getProperty function to determine the posture. This function searches for an object's property in the database (in this case, posture), and returns the string representation for that property if it is found. You can create new tables that extend the representation of an object, which is found in tutorial six. If there is no posture database, then much of the preparatory action may be commented out. Next, we check to see if the agent is standing. If it isn't, we tell the agent to stand by creating and passing an action dictionary (just like we do in the execution step). PAR is, at its heart, a backward chainer. So, we can check and return as many actions as we need to in order to get the agent in the proper state to perform an action. In this case, the agent only requires one step, standing (which we'll define in a bit).

Next, we should define the execution step. Like last time, this is just a primitive action we are telling the agent to perform, and setting the agent to *"Operating"* so other actions can know that the agent is busy. In the execution script, add the following lines:

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

Just like in our basic action, we define a dictionary of actions. In this case, our walk action requires both an agent to walk and an object to walk to, so we need to pass all of that information through the action.

Next, we need to define how we end our action. It would be nice to end the action as soon as the agent reaches the object, and so we can use two more defined python functions to do that. Within the termination script, add:

```
def culmination_condition(self, agent,obj):
        radius = getBoundingRadius(obj);
        distance = dist(agent, obj);
        if distance <= radius:
                setProperty(agent,"obj_status","IDLE");
                return SUCCESS
        else:
                return INCOMPLETE
```

Within this script, we use the getBoundingRadius function to decide a minimum distance to the object. Using this, it is assumed the minimum distance required to be at an object is the bounding radius. Next, we determine the distance between the agent and the object (measured from their given position). When the distance between the two objects is within this radius, we assume the action has finished.

Finally, we may need to update some world information when we finish running the action. Normally, this step is used to deallocate resources that are no longer being used (such as setting agents back to an idle state or saying an object is no longer being held by an agent). For this, we want to make sure the world knows that the agent is now idle (and can be given new actions). To do this, add the following lines to the post-conditions

script:

Now that our walk scripts are created, we need to create scripts in order to stand. However, the applicability and pre-conditions look exactly the same as in the basic tutorial, and so will not be covered here. For the execution step, we simply wish to create a dictionary telling the system to perform the stand action. For the execution script of stand, add the following lines:

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

This action should terminate when the agent is has spent enough time trying to stand, or a signal from the program explains the action is finished. Culmination conditions are also used to determine if the agent should start the action in the first place. In this case, the termination conditions form a double check of the world before the agent begins the action. Therefore, the termination condition also checks to see if the agent is standing. Add the following lines to the termination script:

```
def culmination_condition(self, agent):
        if self.start_time+self.duration > getElapsedTime() and finishedAction(self.id):
                setProperty(agent,"obj_status","IDLE");
                setProperty(agent,"obj_posture","STANDING");
                return SUCESS
        if getProperty(agent,"obj_posture") =="STANDING":
                return SUCCESS
        else:
                return INCOMPLETE
```

Once all of these scripts are defined, the actions can be added to the database, just like in the basic action tutorial. Luckily, these actions have already been added to the database. Remember that the walk action requires an object, and so the act_obj_num should be set to 1.

Now that the scripts are defined, action code can be generated to connect the logic to actions within the world. To allow the agent to stand, the agent needs have its posture property set to stand. In order to do that, we can use the following code:

```
int doStand(iPAR *ipar){
        MetaObject *subject=ipar->getAgent();
        printf("Agent %s is standing up",agent->getObjectName());
        ipar->par->setFinished(true);
        return 1;
}
```

This function simply grabs the agent object, and prints that the agent is getting up. Also, when the agent has finished this action, the finished flag is set to true, so that action can know it's finished. Next, we need to define a walking action. Assuming you already have a method to move agents towards objects, the walking function can simply be:

```
int doWalk(iPAR *ipar){
        MetaObject *subject=ipar->getAgent();
        MetaObject *obj1=ipar->getObject(0);
        WALKING_FUNCTION(subject,obj1);
        return 1;
}
```

Where WALKING_FUNCTION is your already defined function to have the agent move to the object. After adding these functions to the action table, like we did in the basic action tutorial, you can use these actions exactly like the speaking action in the basic tutorial.