Tutorial 4
Advanced Agents

In this tutorial, well look at subclassing the AgentProc class to create our own agents, containing information such as the agent status. Before you begin this tutorial, you should look at tutorial 1 (basic agents) and 2 (basic actions).

Creating an Agent Class

Well start by creating an Agent class that inherits from the AgentProc class. This agent will contain a simple update function (which is a virtual function within the AgentProc class) which will give a default action (in this case, Speak) if the agent is not doing anything. Lets start by creating an agent header file, such as agent.h. Within this file, place the following code:

```
#include "agentproc.h"

// Creating a subclass of AgentProc to customize behaviors
class Agent : public AgentProc {
        public:
                Agent(char* agentName);         // Customized constructor
                int update(void *val);         // Basic update method
                void addNewAction(iPAR *action);    // We extend the addAction
        private:                                    //  method to update agent state
                void addIdleAction();     // A method for initiating an idle behavior
                bool idle;     // Is the agent idle?
                int idleCount;     // Start an idle action only after a period of time
};
```

Now that we have our methods declared, we can flush out these methods. First, our new agent class is going to require a partime variable, so add to an agent.cpp file:

```
extern parTime *partime; // global pointer to PAR time class
```

The first method, the constructor, just allows us to set our new variables. Within your code, place the following:

The next function, update, is really the workhorse of the agent class. Within it, we can determine properties of the agent and add PARs as needed. This method gets called at every time step of the simulation and represents the AI of the agent. In this case, all we and to do is determine if the agent is idle, and if it is, call the addIdleAction method. In the code, add the following lines:

```
int Agent::update(void* val){

        if(!activeAction())     // Are there any PAR actions being processed/performed?
                idleCount++;     // Wait a few frames before starting an idle behavior

        if(idleCount > 5)
                idle = true;

        if(idle)                 // Start the agent's idle behavior
                {
                        std::cout << "Starting idle behavior" << std::endl;
                        addIdleAction();
                        idleCount = 0;
                }
        return 0;
}
```

The first thing we check in the update method is if we have an activeAction (which is defined in the AgentProc Class). If no actions are Active, then we increase the idleCount. Next, we say that if the idleCount is greater than five, the agent is idle (We dont want to immediately jump to an idle action). Lastly, if the agent is idle, we reset the count, and add our IdleAction to the agent.

Next, lets define a way to add actions to our agent. Add the following lines to the code:

```
void Agent::addNewAction(iPAR* ipar){
        addAction(ipar);
        idle = false;
}
```

This method just adds an action using the AgentProc addAction method, and sets idle to false (since we have an action to perform). This is not the most precise method, as adding an action does not mean the agent will start to perform the action immediately, but for the sake of this tutorial, were assuming having an action is close enough to starting an action.

Finally, we can define a default action (Speak). Almost all of this code is the same as adding an instanced PAR in the basic action tutorial.

```
void Agent::addIdleAction(){
        iPAR* iparTest = new iPAR("Speak", this->getName());
        iparTest->setPriority(3);
        this->addNewAction(iparTest);
}
```

Creating this idle action is not much different than creating any other action. We do set the priority to be lower (e.g. 3) so that other actions will preempt this idle behavior. Now, we can use our Agent class just like we used the AgentProc class in the basic agent tutorial. The only thing to remember is to call the Agents update method after (or possibly before) calling the LwNetList::advance() method. If our agent variable is named agent, this would look like:

```
        for(int i=0; i<500; i++){
                agent->update(0);
                LWNetList::advance(&error);
        }
```

Belief Desire Intent Agent

PAR can also be used to create specific kinds of agents, such as BDI agents described by Woolsworth and implemented in several agent programming languages such as AgentSpeak. A BDI agent works by providing the agent with a purpose(desire) and having it examine the world around it (it's belief) and forming an intent, which is synonymous to an instantiated PAR (iPAR). We start by creating an agent method:

```
bool createIntentFromDesire(const char* desire);
```

The first step of a BDI agent is to parse the agent's desire. In PAR, an actions purpose can represent an atomic interval of a desire. In the Actionary, one MetaAction field is *act_purpose_achieve*. To find all MetaActions with a certain purpose, we can use the Actionary method *getAllPurposed*. This useage is seen as:

```
if (desire == NULL)
        return false;
        std::vector<MetaAction*> possible_acts=actionary->getAllPurposed(desire);
        if(possible_acts.empty())
                return false;
```

This will provide all MetaActions that fulfill a certain purpose, regardless of if the actual PAR is defined.

The success or failure of an action is highly dependent on an agent's understanding of the world state (its belief). For atomic actions, the available resources can determine

which actions are more appropriate for a given scene. For our simple agent, if an action does not use any objects or the objects exist in the environment, we will consider the belief space to be valid for that action. We determine appropriate objects for actions through the use of action affordances, defined by the database table *obj_afford*. This table can be queried from:

```
MetaObject* MetaAction->searchAffordances(int position,int which);
MetaAction* MetaObject->searchAffordances(int position,int which);
int Actionary->searchAffordances(MetaObject *obj,MetaAction *act);
```

For our system, we know the actions and object that we want to use, and so can simply query to determine the correct position. At the top of agent.cpp, we should add in a data structure to hold the defined objects (which is not required, but rather convenient and used in the rest of this example) such as:

```
extern std::list<MetaObject*> all_objs;
```

This data structure should also be added into main.cpp, and any created objects should be stored in it. With this data structure, we can naively search the belief state for objects that can be used with actions by:

```
for(std::vector<MetaAction*>::const_iterator it=possible_acts.begin();
        it != possible_acts.end();
        it++){
        int num_objs=(*it)->getNumObjects();
        if(num_objs < 0){
                if(act == NULL)
                        act=(*it);
        }
        else{
                bool satisfied=true;
                for(int i=0; i<num_objs; i++){
                        bool found=false;
                        for(std::list<MetaObject*>::const_iterator o_it=all_objs.begin();
                                o_it != all_objs.end();
                                o_it++){
                                if(actionary->searchAffordance((*it),(*o_it))==i)
                                        found=true;
                        }
                        if(!found)
                                satisfied=false;
                }
                if(satisfied){
                        act=(*it);
                }
        }
}
```

The last part of our BDI agent is intent. For our simplified BDI agent, we consider an intent to be an instantiated PAR that is on the agent's queue to execute. Like in previous examples, our agent can instantiate an action through the following code, which also re-determines the objects the agent should use.

```
if(act == NULL)
        return false;
//Finally, start the intent
iPAR *ipar=new iPAR(act->getActionName(),this->getObject()->getObjectName());
for(int i=0; i<act->getNumObjects(); i++){
        for(std::list<MetaObject*>::const_iterator o_it=all_objs.begin();
                o_it != all_objs.end();
                o_it++){
                if(actionary->searchAffordance(act,(*o_it))==i){
                        ipar->setObject((*o_it),i);
                        break;
                }
        }
}
ipar->setPriority(5);
ipar->setStartTime(partime->getCurrentTime());
ipar->setDuration(5);
this->addAction(ipar);
return true;
```

This is all that is needed to create a simple BDI agent using the PAR system. Purposes can be given to agent, and an action should be generated from that purpose. It is important to examine what was just done in case the agent does not perform an action. Remember that Meta-Actions must be linked with purposes to search over them, and objects must exist in the world for agents to use them. Also, affordances should be defined for each action and position an object (or it's parent) can be used in.