PAR API for SUBTLE MURI

November 2007

1. Creating the Simulation (Objects, Agents, Actions) (See below)
2. Queries
    a. Existence/contents/location of: Objects, Agents, Actions
    b. Properties of Objects, Agents, Actions
    c. States of: Objects, Agents, Actions
    d. Spatial Configurations
3. Modifying Properties (See below)
4. Modifying Behaviors
    a. Add/remove/reorder iPARs on an agent's queue
    b. Change any iPAR parameters
    c. Change the uPARs
5. Histories
    a. Consists of a vector/list/hash of pointers to the ActionTable, which I believe already holds all of the actions that have been queued by an agent.
    b. Building
        i. Depends on data structure.
        ii. Depends on possible queries.
        iii. Could be as basic as pushing the iPAR pointer onto a list.
        iv. Always done automatically?
        v. Able to be modified?
    c. Retrieving
        i. Across agents?
        ii. Based on what parameters?
            1. Time
            2. Action name
            3. Related objects
            4. Rooms/location
            5. Priority
            6. Failures
            7. Origin (result of other PARs or Commander)
6. Failures (See below): Basically an error code and all iPAR values.

Setting up the virtual environment

- Create the geometry for the virtual environment.
- Create PAR objects. Any object in the environment that you want to manipulate with PAR must be in the object hierarchy. Once an object is in the object hierarchy you may want to set some of its properties. Otherwise, it will just inherit them from its parent.
  There are two ways to create PAR objects.

First, you can name the figures in the environment the same name as a node in the object hierarchy of the Actionary followed by an underscore and a number. Then, if you call the update working memory from the environment routine, it will automatically go through the figures in the environment and look for nodes in the object tree with the same name as the portion of the figure name before the underscore.
If the routine finds such a node, it will create a PAR object with the same name as the figure and makes its parent the node it found. For example, if ``Car" is in the object hierarchy and you have a figure in the environment named ``Car_1", update working memory from environment will create a ``Car_1" object and place it in the object tree with ``Car" as its parent. ``Car_1" will then inherit ``Car"s properties.
Second, you can initialize the objects explicitly.

Here is an example of explicitly putting an object in the hierarchy and setting some of its properties:

```
// Find an appropriate node in the object tree
MetaObject *parent = (MetaObject *)objtree->searchByName("Physical");
// Create a Cup node under Physical that will be used as a parent
// for all cups.
objtree->create("Cup", parent, 0);
// Get a pointer to the node we just added.
parent = (MetaObject *)objtree->searchByName("Cup");
// Create an instance of Cup under the Cup node.  There should also be a figure in the environment named
// Cup_1.
objtree->create("Cup_1", parent, 0);
// Get a pointer to the node we just added
MetaObject *obj = (MetaObject *)objtree->searchByName("Cup_1");
// Set a grasp site for the object
obj->addGraspSites("Cup_1.cadobject.base0");
```

Adding agents

- Before creating an agent process, there must already be an object in the hierarchy created with the same name and 1 as the third argument of create (See above).
- Create a subclass of **AgentProc** for your agent, and be sure to write the virtual inform method.
- Create a new agent process:

```
// MyAgent is a subclass of AgentProc
MyAgent *agent = new MyAgent(``ObjectName_1");
```

Adding actions

- First, add the new action to the action tree of the Actionary, and set its properties. This process is similar to adding an object to the object tree:

```
MetaAction *parent = (MetaAction *)acttree->searchByName("NonHumanAction");
MetaAction *movecamera = acttree->create("MoveCamera", parent);
acttree->setNumberOfObjects(movecamera, 0);
```

- Next, make sure that the agents you want to execute the action have the capability to do so.

```
MetaObject *cameraObject = (MetaObject *)objtree->searchByName("camera");
cameraObject->setCapability("MoveCamera");
```

- Then, register the action in the ActionTable:

```
ActionTable::addFunctions("MoveCamera", &move_camera, &camera_done);
```

- Finally, make sure that the proper conditions for the action are setup:

```
acttree->setCulminationCond(movecamera,"/home2/allbeck/research/camera/MoveCameraCul.py",true);
acttree->setApplicabilityCond(movecamera,"/home2/allbeck/research/camera/MoveCameraApp.py",true);
acttree->setPreparatorySpec(movecamera,"/home2/allbeck/research/camera/MoveCameraPre.py",true);
acttree->setExecutionSteps(movecamera,"/home2/allbeck/research/camera/MoveCameraExec.py",true);
```

Action Failure Handling

The PAR architecture is set up to handle failures during the execution of an action. During execution, all failures are detected and reported by motion generators to the process manager within the agent process.

The failure specified by the motion generator contains the following information:
a) Failure code (represented as a string)
b) Pointer to the instantiated PAR unit which failed
c) Relevant information stored in compact structure to help in failure recovery at a later stage.

The process manager does error recovery and handling in two ways:
1) The process manager freezes the queue manager such that no other actions are popped from the highest level of the ipar queue. It also informs the agent process of a failure and passes it the above information. The agent process can now make a decision about the failure recovery process. It could restart the queue as is or by flushing the entire action queue of the agent or by inserting a new high priority action into the queue.
2) The process manager also takes care to abort all actions that lie in the path from the failed primitive action up to the highest complex action (which hierarchically invoked the failed primitive action). i.e., for example, if deep within a high level complex action, a primitive action fails, then all actions up to the complex action are automatically aborted. Alternately, different action(s) can be selected anytime instead of aborting all the actions.

This mechanism allows for a very general approach towards failure detection and recovery. Based on the simulation, the agent process could also invoke the planner to generate a new path or set of actions using the information passed from the motion generator.

- The second call-back function of an action (see **addFunctions**) should be constructed such that when a failure occurs:
  1. *It sets up the FailData argument. FailData contains:*
     a. string failcode
     *b. iPAR *ipar*
     *c. void *data*
  2. *It returns 2, indicating a failure has occurred in the Motion Generator.*
- When the call-back function returns 2, the PAR system will know that a failure has occurred and:
  1. *Halt the agent's queue to prevent it from trying to execute any further actions. (Actions can still be added to the queue, but no new actions will be popped from it???)*
  2. *Call the agent's inform method with ``failure'' as the flag and FailData as its arguments.*
- How failure recovery is handled is left up to the **Agent Process**. It can begin executing actions again by either:
  1. *Just restarting its queue (restartQ();)*

  or

  2. *Flushing all of the actions currently on the queue and restarting it (flushQ();)*

Agent Process

Class: AgentProc	
File: agentproc.h This is the base class for all agent processes. If the agent has been defined in the database, it creates a new queue for instantiated PARs (iPARs) and maintains the action queue of the agent.	
AgentProc(char *agentname);	
Constructor**:** creates a new agent process and internally spawns the AgentNet for monitoring the agent's action queue.	
Parameters:	**agentname**: Name of the agent
int addAction(iPAR *ipar);	
Add an action to the agent's queue. This has to be specifically invoked by the user.	
Parameters:	**ipar**: Pointer to the iPAR that has to be added to the action queue of the agent
int error(void);	
Check if there is an error in the initialization of the constructor. If there is an error, check to see if the agent has been defined in the database.	
virtual void inform(char* flag, void *args);	
A way to pass messages to other agent processes.	
Parameters:	**flag**: Indicates the type of message being sent. Two flags, f1 and f2 can be compared by using their indexes. For example, index[f1] == index[f2].
	args: Any information that needs to be passed to the other agent.

Class: AgentTable	
File: partable.h Maintains a list of all agent processes. A separate agent process is associated with each agent.	
static void addAgent(const char *name, AgentProc *agentproc);	
Add an agent process to the existing list.	
Parameters:	**name**: Name of the corresponding agent.
	agentproc: Pointer to the agent process that is being added. This is invoked internally from AgentProc.
static AgentProc*getAgent(const char *name);	
Get a handle to the agent process with the specified name.	
Parameters:	**name**: Name of the agent

Class: ActionTable

File: partable.h
Maintains a hashed list of all defined actions and their associated call-back functions.

static void addFunctions(const char *name, int (*f1)(iPAR *ipar),
int (*f2)(iPAR *ipar, FailData **data));

Add an action with its associated call-back function to the existing list. This has to be specifically invoked by the user.

Parameters:	
	name: Name of action as it exists in the database of action hierarchy
	f1: Pointer to the call-back function which takes an object of type IPAR as a parameter.
	f2: Pointer to a call-back function which returns 1 to indicate completion of action and returns 0 to indicate action in progress. The function will return 2 if there was an error with the execution of the action. This function is polled internally by the AgentProc class to check for completion of action.

static actfunc *getFunctions(const char *name);

Get the handle to the function for the specified action

Parameters:	
	name: Name of action

Actionary API

Class: MetaAction	
File: metaaction.h PAR, i.e., a node in the action tree.	
MetaAction(char* str);	
Constructor.	
Parameters:	**str**: metaaction name
~MetaAction();	
Destructor.	
void print();	
Print out the node.	
int * objNum;	
Maximum number of objects	

Class: iPAR	
File: workingmemory.h Instantiated PAR.	
iPAR(char* actname, char* agname, char* objnames[]);	
Constructor. Each iPAR contains an uPAR, an agent (performer of the action), and some other objects.	
Parameters:	**actname:** metaaction name
	agname: agent name
	objnames: objects name list
char* print();	
Print out the uPAR, acting agent, and objects	
MetaAction * uPAR;	
Action performed, uninitialized PAR	
MetaObject * agent;	
Acting agent	

MetaObject **objects;
Objects manipulated by agent via action

Class: ActionTree	
File: workingmemory.h Action tree. The working memory of a virtual world can only have a single action tree, which is pointed to by a global variable: ActionTree* acttree;	
ActionTree();	
Construtor.	
MetaAction* create(char* aname, MetaAction* actparent);	
Add a PAR to the tree.	
Parameters:	**aname**: metaaction name
	actparent: pointer to the parent action
bool recycle(MetaAction* act);	
Remove a PAR from the tree.	
Parameters:	**act**: pointer to the action to be recycled
MetaAction* searchByName(char* str);	
Search by name if an action is currently in the action tree. Must be casted using (MetaAction*).	
Parameters:	**str**: name of the action to be searched
char* print();	
Print out all the actions currently in the action tree.	
void addAlias(MetaAction* act, char* name);	
Add an alias to the action pointed by MetaAction* act.	
Parameters:	**act:** pointer to the action.
	name: alias name.
char* searchAlias(MetaAction* act char* name);	
Search if an alias is currently on the alias list of the action pointed by MetaAction* act.	
Parameters:	**act:** pointer to the action to be searched
	name: name of the alias
void setNumberOfObjects(MetaAction* act, int num);	
Set the number of objects a PAR is applied to.	
Parameters:	**act**: pointer to the action
	num: maximum number of objects can be applied by the action

int getNumberOfObjects(MetaAction* act);	
Get the number of objects a PAR is applied to.	
Parameters:	**act**: pointer to the action
int setCondition(MetaAction* act, int which, char* str, bool file = false);	
Set a condition for a PAR. The condition is written in Python script.	
Parameters:	**act**: pointer to the action
	which: condition index: 0 -- applicability condition, 1 -- culmination condition, 2 -- preparatory condition, 3 -- execution steps
	str: the string which contains the Python script
	file: indicates if the Python script is contained in a file or not, default is False
PyObject* testCondition(iPAR* ipar, int which);	
Test if the iPAR satisfies the condition.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
	which: which condition to test. (condition index: 0 -- applicability condition, 1 -- culmination condition, 2 -- preparatory condition, 3 -- execution steps)
int setApplicabilityCond(MetaAction* act, char* str, bool file = false);	
Set the applicability condition for a PAR.	
Parameters:	**act**: pointer to the action
	str: the string which contains the Python script
	file: Python script contained in a file or not, default is False
PyObject* testApplicabilityCond(iPAR* ipar);	
Test the applicability condition of an iPAR.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
int setCulminationCond(MetaAction* act, char* str, bool file = false);	
Set the culmination condition for a PAR.	
Parameters:	**act**: pointer to the action
	str: the string which contains the Python script
	file: Python script contained in a file or not, default is False
PyObject* testCulminationCond(iPAR* ipar);	
Test the culmination condition of an iPAR.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
int setPreparatorySpec(MetaAction* act, char* str, bool file = false);	
Set the preparatory specification for a PAR.	
Parameters:	**act**: pointer to the action
	str: the string which contains the Python script
	file: Python script contained in a file or not, default is False

PyObject* testPreparatorySpec(iPAR* ipar);	
Return the preparatory specification of an iPAR. Note that the specification is written in Python script, so this method returns a pointer to a Python object.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
int setExecutionSteps(MetaAction* act, char* str, bool file = false);	
Set the execution steps for a PAR.	
Parameters:	**act**: pointer to the action
	str: the string which contains the Python script
	file: Python script contained in a file or not, default is False
PyObject* getExecutionSteps(iPAR* ipar);	
Return the execution steps of an iPAR.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
PyObject* testExecutionSteps(iPAR* ipar);	
Return the execution steps of an iPAR. Note that the specification is written in Python script, so this method returns a pointer to a Python object.	
Parameters:	**ipar**: pointer to iPAR. iPAR must be initialized with action, agent and objects.
int setTerminationCond(MetaAction* act, char* str, bool file = false);	
Set the termination condition for a PAR.	
Parameters:	**act**: pointer to the action
	str: the string which contains the Python script
	file: Python script contained in a file or not, default is False
PyObject* getTerminationCond(iPAR* ipar);	
Return the termination condition of an iPAR.	
Parameters:	**ipar:** pointer to iPAR. iPAR must be initialized with action, agent and objects.
PyObject* testTerminationCond(iPAR* ipar);	
Return the termination condition of an iPAR. Note that the specification is written in Python script, so this method returns a pointer to a Python object.	
Parameters:	**ipar:** pointer to iPAR. iPAR must be initialized with action, agent and objects.

Class: MetaObject

File: metaobject.h
Object node in the object tree. Note that objects include not only agents but also physical objects such as a chair, a mug, and so on.

MetaObject(char* obName, bool flag = false);	
Construtor	
Parameters:	**obName**: object name
	flag: indicates if MetaObject is an object or an agent.
~{}MetaObject();	

Destructor.	
char* getObjectName();	
Return the name of the current object.	
Vector<3>* getPosition();	
Get the current Position of the object.	
void setPosition(Vector<3>* vector);	
Set the Position of the object.	
Parameters:	**vector**: vector value
Vector<3>* getVelocity();	
Get the current Velocity of the object.	
void setVelocity(Vector<3>* vector);	
Set the Velocity of the object.	
Parameters:	**vector**: vector value
Vector<3>* getAcceleration();	
Get the current Acceleration of the object.	
void setAcceleration(Vector<3>* vector);	
Set the Acceleration of the object.	
Parameters:	**vector**: vector value
Vector<3>* getForce();	
Get the current Force of the object.	
void setForce(Vector<3>* vector);	
Set the Force of the object.	
Parameters:	**vector**: vector value
Vector<3>* getTorque();	
Get the current Torque of the object.	
void setTorque(Vector<3>* vector);	
Set the Torque of the object.	
Parameters:	**vector**: vector value
Vector<3>* getOrientation();	
Get the current Orientation of the object.	

void setOrientation(Vector<3>* vector);	
Set the Orientation of the object.	
Parameters:	**vector**: vector value
char* getCoordinateSystem();	
Return the site name where the current object is.	
void setCoordinateSystem(char* coord);	
Set the coordinate system of the object.	
Parameters:	**coord**: site name.
STATUS_TYPE getStatus();	
Return the current status of the object.	
void setStatus(char* type);	
Set the Status of the object.	
Parameters:	**type**: status type: 0 -- IDLE, 1 -- MOVING
POSTURE_TYPE getPosture();	
Return the current posture of the object.	
void setPosture(char* type);	
Set the Posture of the object.	
Parameters:	**type**: posture type: 0 -- NEUTRAL, 1 -- SIT, 2 -- STAND, 3 -- PRONE, 4 -- SUPINE, 5 -- KNEEL, 6 – OPEN, 7 -- CLOSE
void addGraspSites(char* siteName);	
Add one site to the grasp site list.	
Parameters:	**siteName**: site name.
list<char*>* getGraspSites();	
Return the list of grasp sites	
bool searchGraspSites(char* siteName);	
Search if one grasp site is on current object's grasp site list.	
Parameters:	**siteName**: site name.
void addContents(MetaObject* obj);	
Add one object to current object's content list.	
Parameters:	**obj**: pointer to the MetaObject to be added.
list<char*>* getContents();	

Return the content list.	
bool searchContents(char* objName);	
Search if one MetaObject is on THIS object's content list.	
Parameters:	**objName**: MetaObject name.
ActApp* searchCapability(char* action);	
Search to see if the action can be performed by the agent (if this object is an agent), or to the physical object (if this object is a physical object).	
Parameters:	**action**: action name
int setCapability(char* action, char* applicability = NULL, bool file = false);	
Add an action to the capability of the object.	
Parameters:	**action**: action name
	applicability: pointer to the applicability, either is a Python script string or a Python script file
	file: indicates if the applicability is contained in a Python script file or not. The default is False.
bool removeCapability(char* action);	
Remove an action from the capability of the object.	
Parameters:	**action:** action name
bool containObject(MetaObject* obj);	
Check if an object is in the content of this object.	
Parameters:	**obj:** pointer to the MetaObject checked
void print();	
Print out the object.	

Class: ObjectTree

File: workingmemory.h
Object tree. The working memory of a virtual world can only have a single object tree, which is pointed to by a global variable: ObjectTree* objtree;

ObjectTree();	
Constructor.	
MetaObject* create(char* objname, MetaObject* objparent, bool agent = false);	
Add an object to the tree.	
Parameters:	**objname**: object name
	objparent: the pointer to the parent object
	agent: indicates if the object created is an Object or an Agent.

bool recycle(MetaObject* obj);	
Remove an object from the tree.	
Parameters:	**obj**: pointer to the object to be recycled
MetaObject* searchByName(char* str);	
Search by name if an object is currently in the object tree. Must be cast using (MetaObject*).	
Parameters:	**str**: name of the object to be searched
char* print();	
Print out all the objects currently in the object tree.	
void* getProperty(MetaObject* obj, char* pname);	
Get a property of an object. Note that objects inherit properties from their ancestors.	
Parameters:	**obj**: object name
	pname: property name
bool setProperty(MetaObject* obj, char* pname, void* pvalue = NULL);	
Set a property of an object.	
Parameters:	**obj**: object name
	pname: property name
	pvalue: property value, in generic type, can be casted to/from any data type, but must be consistent.
bool deleteProperty(MetaObject* obj, char* pname);	
Delete a property of an object.	
Parameters:	**obj**: object name
	pname: property name
ActApp* searchCapability(MetaObject* obj, char* action);	
Search to see if the action is in the capability of an object.	
Parameters:	**obj**: object name
	action: action name
void move(MetaObject* obj, MetaObject* where);	
Move one object into another. This accomplishes setLocation, addContents, and deleteContents	
Parameters:	**obj**: pointer to the source MetaObject
	where: pointer to the destination MetaObject
void setStdVector(MetaObject* obj, int which, Vector<3>* vector);	
Set one of standard properties for the object.	
Parameters:	**obj**: pointer to the MetaObject
	which: standard vector index (which: 0 -- Position, 1 -- Velocity, 2 -- Acceleration, 3 -- Force, 4 – Torque 5 -- orientation)
	vector: vector value

Vector<3>* getStdVector(MetaObject* obj, int which);	
Get one of standard properties of the object.	
Parameters:	**obj**: pointer to the MetaObject
	which: standard vector index(which: 0 -- Position, 1 -- Velocity, 2 -- Acceleration, 3 -- Force, 4 – Torque, 5 -- orientation)
	vector: vector value
int setPyMethod(MetaObject* obj, char* mname, char* mstr, bool file = false);	
Define a method for the object node by using Python script. Note that like properties, methods are also inherited by children nodes.	
Parameters:	**obj**: pointer to the MetaObject
	mname: python method name
	mstr: the string which contains the Python method
	file: Python script contained in a file or not, default is False
PyObject* invokePyMethod(MetaObject* obj, char* mname, char* args);	
Invoke a method of the object node.	
Parameters:	**obj**: pointer to the MetaObject
	mname: python method name
	args: arguments used upon invocation