Overview of the MCL API
Matt Schmill
5/4/2009

Modes of operation

MCL may be used in three different ways. For C/C++ code, the library may be linked directly to the host executable and functions in the mclMA namespace may be interleaved with host code. For Java code a JNI interface is available. For all other systems, a TCP/IP server can be built and the API functions accessed using a plaintext protocol.

Step 1: Initialization

A single instantiation of MCL may monitor many host agents. Each agent must register with MCL using a unique key of type string. Initialization is performed using the following API function.

mclMA::initializeMCL(string key, int Hz) *initialize(mars_rover,0)*
Key is a unique string identifier for the host agent.
Hz is a limiting frequency on MCL monitoring. A value of 0 specifies no limit.

All API calls will be specified in this document as above. The name in bold is the C++ library function signature and the italicized version is an example using the TCP/IP plaintext function format.

Once the key is initialized, the MCL instantiation associated with it should be configured. Configuration consists mainly of loading default parameters for the system including the Bayes priors for the ontologies. Configuration is accomplished as follows.

mclMA::configureMCL(string key,string dom) *initialize(mars_rover,mars)*
mclMA::configureMCL(string key,string dom,string ag) *initialize(mars_rover,mars,rover)*
mclMA::configureMCL(string key,string dom,string ag,string con) *initialize(mars_rover,mars,rover,ugpa)*
Key is a unique string identifier for the host agent.
Dom identifies the domain.
Ag identifies the agent type.
Con identifies the controller type.

The configure function attempts to find configuration files using $MCL_CORE_PATH and appending the supplied domain, agent, and controller (if they are supplied) as subdirectories. That failing it will attempt to use $HOME and the cwd. That failing, it will use system defaults. Be sure to set $MCL_CORE_PATH if you want to use defaults that come with the distribution.

Step 2: sensor specification

Once configured, the host should specify its sensors using the mclMA::observables API. This allows the host to specify observables in two ways: as sensors native to the host agent, or as observable properties assigned to external objects in the world. Many agents will utilize both. Self-observables are the simplest, and are defined as follows:

using namespace mclMA::observables;
declare_observable_self(string key,string name,double def) declareObservableSelf*(mars_rover,fuel,100)*
Key is a unique string identifier for the host agent.
Name is an observable identifier unique to the agent.
Def is the default value.

Once defined, a sensor's characteristics may be specified. This process includes specifying properties of the sensor (see APICodes.h), legal values or ranges for the sensor, and noise properties.

using namespace mclMA::observables;
set_obs_prop_self(string key,string name,spkType prop,spvType val)
setObsPropSelf(mars_rover,fuel,property_sensorclass,sensorclass_resource)
Key is a unique string identifier for the host agent.
Name refers to the self observable.
Prop is the property code (see PROP_xxx in APICodes.h)
Val is the property value for this sensor (see xxx_yyy in APICodes.h)
note that currently the TCP/IP interface does not translate strings to the appropriate key/code

using namespace mclMA::observables;
set_obs_noiseprofile_self(string key,string name,spvType profiletype)
set_obs_noiseprofile_self(string key,string name,spvType profiletype,double param)
setObsNoiseProfileSelf(mars_rover,fuel,perfect)
setObsNoiseProfileSelf(mars_rover,fuel,uniform,0.10)
Key is a unique string identifier for the host agent.
Name refers to the self observable.
Profiletype is a coded noise profile specifier.
Param is a parameter required by some noise profiles.

using namespace mclMA::observables;
add_obs_legalval_self(string key,string name,double lval)
addObsLegalValSelf(mars_rover,fuel,99)
Key is a unique string identifier for the host agent.
Name is an observable identifier unique to the agent.
Lval is the legal value to be added to the set of legal values for this observable.

using namespace mclMA::observables;
set_obs_legalrange_self(string key,string name,double min,double max)
setObsLegalRangeSelf(mars_rover,fuel,0,100)
Key is a unique string identifier for the host agent.
Name is an observable identifier unique to the agent.
Min is the lower legal bound for this observable.
Max is the upper legal bound for this observable.

Self observables might include internal state variables or sensors pertaining to the host agent's own operation. In many cases, a fixed vector of simple observables is sufficient. More sophisticated agents have a notion of external objects and their associated properties. These objects may enter and exit the perception of the agent, and as such a fixed vector is inappropriate. In this case, MCL provides functionality for defining *object types* that have their own set of observable properties which can be defined and characterized in the same ways as self observables. Once an object type has been defined, the agent can *notice* instantiations of that type, and those observables will be added to the domain of MCL's monitoring capabilities. The following functions can be used to define, characterize, create, and destroy observable objects and their instantiations.

using namespace mclMA::observables;
declare_observable_object_type(string key,string name)
declareObservableObjType(mars_rover,crater)
Key is a unique string identifier for the host agent.
Name is an object type identifier unique to the host agent.

using namespace mclMA::observables;
declare_object_type_field(string key,string objname,string obsname)
declareObjField(mars_rover,crater,radius)
Key is a unique string identifier for the host agent.
Objname refers to the object type.
Obsname is a field identifier unique to the named object type.

using namespace mclMA::observables;
set_obs_prop(string key,string typename,string fieldname,spkType prop,spvType val)
setObsProp(mars_rover,crater,radius,sprop_sensorclass,sclass_spatial)
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Fieldname refers to a type in the named field.
Prop is the property code (see PROP_xxx in APICodes.h)
Val is the property value for this sensor (see xxx_yyy in APICodes.h)
* *note that currently the TCP/IP interface does not translate strings to the appropriate key/code*

using namespace mclMA::observables;
set_obs_noiseprofile(string key,string typename,string fieldname,spvType profiletype)
set_obs_noiseprofile(string key,string typename,string fieldname,spvType profiletype,double param)
setObsNoiseProfile(mars_rover,crater,radius,np:perfect)
setObsNoiseProfile(mars_rover,crater,radius,np:uniform,0.10)
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Fieldname refers to a type in the named field.
Profiletype is a coded noise profile specifier.
Param is a parameter required by some noise profiles.
* *note that currently the TCP/IP interface does not translate strings to the appropriate key/code*

using namespace mclMA::observables;
add_obs_legalval(string key,string typename,string fieldname,double lval)
addObsLegalVal(mars_rover,crater,radius,100)
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Fieldname refers to a type in the named field.
Lval is the legal value to be added to the set of legal values for this observable.

using namespace mclMA::observables;
set_obs_legalrange(string key,string name,double min,double max)
setObsLegalRange(mars_rover,fuel,0,100)
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Fieldname refers to a type in the named field.
Min is the lower legal bound for this observable.
Max is the upper legal bound for this observable.

using namespace mclMA::observables;
notice_object_observable(string key,string typename,string objname)
noticeObj(mars_rover,crater,crater-1)
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Objname is an object instantiation identifier unique to the host agent.

using namespace mclMA::observables;
notice_object_unobservable(string key,string typename,string objname)
un*noticeObj(mars_rover,crater,crater-1)*
Key is a unique string identifier for the host agent.
Typename refers to an object type.
Objname is an object instantiation identifier unique to the host agent.

Step 3: HIAs

MCL allows the host to initiate anomaly processing directly (without using expectations & monitoring) through the use of Host Initiated Anomalies (HIAs). They must be specified in advance.

using namespace mclMA::HIA;
registerHIA(string key,string HIAname,string activateNode)
registerHIA(mars_rover,noRoute,modelError)
Key is a unique string identifier for the host agent.
HIAName is a unique string identifier for the host agent.
activateNode names an ontology node to activate when the HIA is triggered.

using namespace mclMA::HIA;
signalHIA(string key,string HIAname)
signalHIA(string key,string HIAname,resRefType referent)
signalHIA(mars_rover,noRoute)
signalHIA(mars_rover,noRoute,0x03ef710f)
Key is a unique string identifier for the host agent.
HIAName is a unique string identifier for the host agent.
activateNode names an ontology node to activate when the HIA is triggered.

The HIA provides a direct route to anomaly processing in MCL by attaching an initial activation pattern[1] for the ontologies with the HIA name. The HIA may be signaled with or without a referent. Referents are described below.

Step 4: Property vectors

Property vectors allow the host agent to indicate to MCL what some or all of its capabilities include. MCL, when processing an anomaly and configuring the Bayes nets, may query the property vector to better compute priors. For example, an agent using a policy learned by reinforcement learning would want MCL to keep RL-appropriate responses under consideration, but it might be fruitless for MCL to consider model-based responses if none existed in the host. MCL works off of a default property vector and allows additional vectors to be generated and used where appropriate. Here we will consider the API functions for modifying the default property vector.

Note that the property vector stuff seems to need revision.

using namespace mclMA;
setPropertyDefault(string key,pkType property,pvType value)
setPropertyDefault*(mars_rover,property:sensors_can_fail,propcode:true)*
Key is a unique string identifier for the host agent.
Property is an encoded property key (see APICodes.h)
Value is a property value (encoded or not).
* *note that currently the TCP/IP interface does not translate strings to the appropriate key/code*

using namespace mclMA;
newDefaultPV**(string key,string HIAname)**
signalHIA(mars_rover,noRoute)
Key is a unique string identifier for the host agent.

using namespace mclMA;
popDefaultPV**(string key,string HIAname)**
signalHIA(mars_rover,noRoute)
Key is a unique string identifier for the host agent.

The *new* and *pop* functions provide the host with a way to create and remove the default property vector when major state changes occur in the system. This is not necessary under most circumstances, as you will see in the next section, as MCL provides a way to produce local copies of the default property vector that can be modified in a context-dependent way.

---

1 Currently only a single node is associated with the HIA but this is planned to be expanded to sets of nodes.

Step 5: Expressing expectations

Once the system, its sensors, properties, and HIAs have been initialized, MCL is ready to be incorporated into the normal monitor-and-control loop of the host. Each time the host agent engages in an activity about which there are some expectations, the host should open an *expectation group* and attach to it one or more *expectations*. Expectation groups are provided to organize expectations around activities, and have local property vectors and parent pointers so that hierarchical activity can be expressed to MCL.

There are many types of expectations that can be maintained and monitored by MCL. See APICodes for a listing of the available expectation types (EC_xxx defines). The following functions are provided for creating expectation groups and expectations, and for closing expectation groups.

using namespace mclMA;
declareExpectationGroup**(string key,egkType group_key)**
declareExpectationGroup(mars_rover,0x00000001)
Key is a unique string identifier for the host agent.
group_key is a unique (unsigned int) identifier for the expectation group.

using namespace mclMA;
declareExpectationGroup**(string key,egkType group_key,egkType parent_key,resRefType referent)**
declareExpectationGroup(mars_rover,0x00000002,0x00000001,NULL)
Key is a unique string identifier for the host agent.
group_key is a unique (unsigned int) identifier for the expectation group.
parent_key is an (unsigned int) that is the expectation group's parent's group key.
Referent refers to an MCL frame (see the section on monitoring).

using namespace mclMA;
declareExpectation**(string key,egkType group_key,ecType code,float arg)**
declareSelfExpectation**(string key,egkType group_key,ecType code,string self_sensor)**
declareSelfExpectation**(string key,egkType group_key,ecType code,string self_sensor,float arg)**
declareObjExpectation**(string key,egkType group_key,ecType code,string obj,string obs)**
declareObjExpectation**(string key,egkType group_key,ecType code,string obj,string obs,float arg)**
declareDelayedExpectation**(string key,double delay,egkType group_key,ecType code,string slf_sensor)**
declareDelayedExpectation**(string key,double delay,egkType group_key,ecType code,string slf_sensor,float arg)**
declareExp(mars_rover,0x00000001,ec_realtime,5.0)
declareSelfExp(mars_rover,0x00000001,ec_nonetchange,velocity)
declareSelfExp(mars_rover,0x00000001,ec_stayunder,velocity,5.0)
declareObjExp(mars_rover,0x00000001,ec_nonetchange,enemy,velocity)
declareObjExp(mars_rover,0x00000001,ec_stayunder,enemy,velocity,5.0)
declareDelayedExp(mars_rover,0x00000001,2.0,ec_maintainvalue,velocity)
declareDelayedExp(mars_rover,0x00000001,2.0,ec_stayunder,velocity,5.0)
Key is a unique string identifier for the host agent.
group_key refers to an existing expectation group.
Code is a expectation code that specifies the type of expectation (see mcl_symbols.h).
arg is an argument to the expectation constructor.
self_sensor specifies which sensor to monitor, if the code refers to a monitoring expectation.
Delay specifies the time, in seconds, to wait before activating the expectation.

using namespace mclMA;
expectationGroupAborted**(string key,egkType group_key)**
expectationGroupAborted(mars_rover,0x00000001)
Key is a unique string identifier for the host agent.
group_key refers to an existing expectation group.

using namespace mclMA;
expectationGroupComplete**(string key,egkType group_key,observables::update& the_update)**
expectationGroupComplete(mars_rover,0x00000001,{x=1, y=2, ... ,velocity=0})
Key is a unique string identifier for the host agent.
group_key refers to an existing expectation group.
the_update is variously an observables::update object including updates on all sensor and object property values (C/C++), or is a brace-delimited, comma-separated textual update of all sensor and object property values (TCP).

When perusing the types of expectations (the EC_xxx codes) you will note that there are two major categories of expectations: maintenance expectations and effects. Maintenance expectations will be checked at the monitor frequency of MCL and must always pass their associated tests as long as the expectation group is active. Effects will only be checked when the expectation group is declared complete. This is why a sensor update is required during the expectationGroupComplete call – so that the effects expectations can be checked against the most recent sensor values. More info on violations will be provided in the next section.

Step 6: Monitoring

With all of the declarations and initialization steps taken care of, the host system should interleave synchronization steps with its own monitor and control functionality. The monitor() api function should be inserted into the control loop of an agent in such a place that it will be called at least at the desired monitor frequency. The monitor() api function returns a responseVector object (which is a vector of mclMonitorResponse objects) and the TCP version will return a textual representation of the responseVector.

The monitor function (as well as the updateObservables API function) accepts an *update* object to keep MCL coordinated with the latest observed values for all observables. It is essentially a lookup table of observable-value pairs. In the TCP versions, a textual representation is sent.

using namespace mclMA;
updateObservables**(string key,observables::update& update)**
updateObservables(mars_rover,{x=1, y=2, ... ,velocity=0})
Key is a unique string identifier for the host agent.
Update is an update object (C/C++) or a textual representation (TCP).

using namespace mclMA;
monitor**(string key,observables::update& update)**
monitor(mars_rover,{x=1, y=2, ... ,velocity=0})
Key is a unique string identifier for the host agent.
Update is an update object (C/C++) or a textual representation (TCP).

It is worth noting that *off-cycle* effect violations (those which are noted outside of a monitor() call) will be returned by the next monitor() call.

The interface to the monitorResponse object is found in mclMonitorRepsonse.h. Essentially the response code must be checked against the RC_xxx codes in APICodes.h and it is up to the host to execute MCL's recommendations and provide feedback.

Step 7: Providing Feedback

The final step to integrating MCL with a host agent is to provide feedback when MCL detects anomalies and makes recommendations. Feedback can either be explicit or implicit. Implicit feedback is given when an agent uses a *referent* during normal operation. A referent is a void* included in a monitorResponse that allows the host to indicate that some action(s) it is taking are part of the course of action suggested by MCL. Referents can be provided when expectationGroups are created.

Explicit feedback allows the host to indicate to MCL that it is accepting or ignoring its suggestions, or in the case of interactive suggestions, the result of the suggested course of action. Explicit feedback is especially important when the repairs suggested by MCL are not themselves part of what is being monitored by MCL.

using namespace mclMA;
suggestionImplemented**(string key,resRefType referent)**
suggestionImplemented(mars_rover,0x3fe477c8)
Key is a unique string identifier for the host agent.
Referent is a 32-bit integer that has been issued in a mclMonitorResponse.

using namespace mclMA;
suggestionIgnored**(string key,resRefType referent)**
suggestionIgnored(mars_rover,0x3fe477c8)
Key is a unique string identifier for the host agent.
Referent is a 32-bit integer that has been issued in a mclMonitorResponse.

using namespace mclMA;
suggestionFailed**(string key,resRefType referent)**
suggestionImplemented(mars_rover,0x3fe477c8)
Key is a unique string identifier for the host agent.
Referent is a 32-bit integer that has been issued in a mclMonitorResponse.

using namespace mclMA;
provideFeedback**(string key,bool feedback,resRefType referent)**
provideFeedback(mars_rover,true,0x3fe477c8)
Key is a unique string identifier for the host agent.
Feedback is a true or false (bool) answer to MCL's feedback request.
Referent is a 32-bit integer that has been issued in a mclMonitorResponse.

Appendix: Symbols

In order to support a more portable, readable TCP/IP protocol, MCL implements an automated symbol generation module for specifying constants that provides #define headers for necessary system enum types as well as a symbolic lookup table for use with textual representations of those constants.

Symbols are defined in the mcl/utils/symbols.def file. This flat text file is a proprietary format. Here is what you need to know to understand the file and how to add to it:

lines containing only c-style comments will be copied directly to the mcl-sensors.h file.

def <*SYMBOL*> <*VALUE*>
generates a #define <SYMBOL> <VALUE> in mcl-sensors.h

prefix <*PRE*>
starts a counter associated with PRE, internal to the symbol processor

psym <*SYMBOL*> <*PRE*>
creates a #define PRE_SYMBOL and defines it as the current counter value for PRE, advancing that counter by 1

size <*SYMBOL*> <*PRE*>
creates a #define PRE_SYMBOL and sets it to the current counter value without advancing it

resume <*PRE_N*> <*PRE_O*>
starts a new counter for PRE_N, that starts at the current value of PRE_O

header <*...*>
inserts ... directly into the mcl-sensors.h file

In addition, each symbol generated using *def* or *psym* is inserted into a lookup table created in the source file mcl-symbols.cc The symbol generated is a lowercase version of the define name. So,

def MY_CONSTANT 1001

would generate a lookup entry for “my_constant” that evaluates to 1001 when the following function is called:

metacog::symbols::smartval_int(“my_constant”,&success);

where success is a bool* which will be false if there is no lookup entry for the string supplied or if it does not successfully specify an integer according to the clib function atoi().

The code and header generated in this process are sandwiched inside the files in mcl/utils/symbols.*.start and symbols.*.end, where * is “header” or “source”. Additional code may be added to these files if absolutely necessary.