MCL Integration Guide
11/4/2009
Matt Schmill

This document details the process of integrating MCL into a host system. In the first section, the process is described in general terms, considering how a system might be built from the ground up with MCL integration as a part of the design and implementation process. In the following section, we consider integration using a wrapper-based paradigm in which MCL is implemented in a non-intrusive way. We consider the Mars/Mars Rover domain as a case study in the wrapper-based deployment of MCL.

In the following text, MCL functions will be referred to by their name in the C++ API. Refer to the document "Overview of the MCL API" for their corresponding TCP/IP API command names.

Integrated Host/MCL Development

Deploying MCL in a host system can be broken up into three phases: *initialization*, *expectations & monitoring*, and *anomaly-handling*.

Initialization

It is expected that most host agents will already have an initialization stage into which the MCL start up requirements can be inserted. MCL supports multiple agents, and the following prescription must be repeated for any distinct agent that will be accessing MCL functionality. The necessary steps for initializing MCL are:

1. `initializeMCL(string key, int hz)`
   Generates an instance of a MetaCognitive Loop, stored by the key specified. Each agent must use a unique key! The hz argument can be used to ensure that monitoring does not exceed a specified frequency. If hz is 0, then MCL will run in "synchronous mode".
2. `chooseOntology(string key, string ontology)`
   Tells MCL which ontology file to use for this agent. See "Overview of the MCL ODL" for details on the Ontology Description Language. MCL searches $MCL_CONFIG_PATH/netdefs/<ontology>.ont for a working ontology file. If that does not exist, it checks the global setting "mcl.configpath" for a path to a valid ontology file. For more details on settings, see the document "UMBC_Utils User Guide".
   Should no valid ontology file exist as named, MCL will either throw an exception (if the setting "mcl.doNotFail" is false), or will use the internal function buildOntologies to create default ontologies.
3. `configureMCL(string key,`
   `string domain,[string agent],[string controller])`
   Tells MCL the domain name, agent type, and controller type using the specified key. These identifiers are used to load default probabilities when the Bayes nets are initialized in anomaly handling.
   MCL searches the location $MCL_CONFIG_PATH/config/<domain>/<agent>/<controller> for cp_tables.mcl to acquire the priors. If it does not find it, it will peel off <controller>, then <agent>, and finally it will attempt to get priors from $MCL_CONFIG_PATH/config/default/

4. *observable declarations*
   All self-observables and object definitions should be declared once MCL is initialized and configured. Host designers should be as thorough as possible here, as observables form the basis for expectations, monitoring, and the processing of indications. Observable properties, legal values, and noise profiles should be specified here. For more details on the observable API, refer to "Overview of the MCL API".
5. *Property vector defaults*
   MCL uses a stack of property vectors to manage ongoing agent properties. Properties comprise a set of features relevant to MCL processing that describe the host's capabilities and characteristics. At this stage, the "default" PV – that which resides at the base of the stack – should be populated with relevant property settings. Later on, new property vectors that are issued will be initialized using this property vector, so only properties that are system-wide should be set at this step.
6. `updateObservables(string key,observables::update& the_update)`
   At this point, MCL should be sent an initial update of the existing observables' values. This is accomplished using the "update" class in the C++ API or by constructing an update string if using the TCP/IP API. This step will allow MCL to hit the ground running when the next phase of operation begins.

Expectations & Monitoring

MCL works by maintaining expectations about what should happen as a host system runs. The host system is responsible for declaring its own expectations about its behavior in terms of changes to its observables, and asking MCL to monitor those expectations at appropriate opportunities.

It is appropriate to associate host expectations with the things it does. For typical autonomous agent, expectations might be associated with its control units – closed-loop controllers, actions, plans, policies, and so on. As the agent activates its various control systems (whatever they may be), expectations are posted to MCL. For computational systems (perhaps a natural language agent) that may not have explicit actions or controllers, expectations may be associated with individual computational processes or any segment of logic that has clearly defined inputs, outputs, and possibly well-define intermediate states.

Expectations in MCL can be grouped into *expectation groups*. Each control unit or well-defined computational process should define an expectation group to which individual expectations can be attached. Each expectation group has a key associated with it that is currently of the C type *unsigned long int.* The host is responsible for generating its own keys – one may choose to use an internal counter, hash keys, or pointers to handle this task. MCL uses this key primarily for bookkeeping, though it should be known that MCL may use the expectation group key in trying to resolve whether an anomaly is a recurrence of something it has already seen.

Expectation groups are declared with the function

```
declareExpectationGroup(string key,egkType eg_key,egkType parent_key,resRefType ref);
```

The argument parent_key allows the host to express hierarchical structure, if it exists. For instance, a plan may have an expectation group associated with it, and each individual step (action) may also have an expectation group key. The individual steps can make use of the parent_key parameter to indicate to MCL the hierarchical nature of the groups in play. If there is no parent, NULL (0x0) can be used. The

parameter ref will be explained in the next section (*Anomaly Handling*).

Once an expectation group has been declared, individual expectations can be attached to it. Expectations are declared using the `declareSelfExpectation` and `declareObjExpectation` API functions. These two functions are used to specify expectation about the agents own observables or about the properties of an observable object. There are various signatures for these functions that allow for different kinds of expectation types to be expressed with any necessary arguments. For more information on the specific arguments, and how to express the various types of expectations, see the API Overview document and the API Symbols document.

Once expectations have been attached to an expectation group, they can be monitored by MCL in one of two ways. Agents whose actions or computational processes are modeled as discrete and/or instantaneous should be declared and monitored using MCL's effects-based expectations. These expectations are checked only once, after the action or computational process has completed. Agents whose actions are durative and continuous should be declared and monitored using MCL's maintenance-based expectations. Consider the following code, adapted from the discrete Mars Rover simulation:

```
1. declareExpectationGroup("rover",&my_moveAction);
2. declareSelfExpectation("rover",&my_moveAction,"location",EC_TAKE_VALUE,dest);
3. my_moveAction->execute();
4. load_with_sensor_values(mcl_update_object);
5. declareExpectationGroupComplete("rover",&my_moveAction,mcl_update_object);
6. responseVector mcl_rv = monitor("rover",mcl_update_object);
```

This is a simplified version of how to do integrated MCL deployment in discrete agent simulation using effects-based expectations. First, declare an expectation group using the pointer to the action (again, you could generate an unsigned long int any way you want). Next, declare an expectation on the action, in this case using the code EC_TAKE_VALUE, which means the observable "location" should take the specified value dest when the action is complete. In line 3, the action code executes. Line 4 loads an update object (class metacog::observables::update) with the agent's current observable values. Line 5 declares the expectation group complete. Note that an update object should be passed to the expectation group completion call so that MCL can lock the sensor values in place at the time of completion. This is because expectations are not monitored when declareExpectationGroupComplete() is called. All expectation monitoring is handled within the call to monitor(). Effects-based expectations are merely *marked* for effects-based monitoring during the group complete call. In line 6, the monitor function is called and a responseVector is returned. The responseVector will summarize any anomalies found by MCL, or it will be empty if there are none.

Monitoring in continuous systems is slightly different, though the concept is the same. Consider the following code, adapted from the RonCon robot control architecture:

```
void generic_controller::execute() {
  declareExpectationGroup("robot",this);
  declareSelfExpectation("robot",this,"battery",EC_STAYOVER,12.7);
  while (keep_going()) {
    do_control_stuff();
    load_with_sensor_values(mcl_update_object);
    responseVector mcl_rv = monitor("robot",mcl_update_object);
    if (!mcl_rv.empty()) {
       deal_with_problem(mcl_rv);
    }
  }
  declareExpectationGroupComplete("rover",&my_moveAction,mcl_update_object);
}
```

The primary difference here is that the monitor() function is called repeatedly inside the control loop to ensure that expectations are maintained throughout the activity. The expectation group is declared complete only when the controller stops.

It is worth noting that when working with a continuously monitored agent architecture, effects expectations can be used. They will be ignored by MCL inside the control loop but they will be considered when the expectation group is declared complete. In this model, the monitor() call immediately following the expectationGroupComplete() call will report any expectation violations.

It is also of note that expectation groups can be aborted, in which case effects expectations will *not* be checked on the way out. This is useful when the host is able to determine an anomaly has occurred on its own, and will expect a violation. The host may choose to skip MCL's processing of the expected violation, and use expectationGroupAborted() instead of the Complete() function.

Anomaly Handling

The monitor() call returns a structure of type responseVector. It is the responsibility of the host to digest the information in the responseVector and inform MCL of what has been done. This is the process of anomaly handling.

The responseVector type is simply a vector of pointers to mclMonitorResponse objects. Each object in the vector will correspond to what MCL considers a single anomaly. An anomaly might represent one or more individual expectation violations. The mclMonitorResponse class serves as the root of a hierarchy of response classes. Any subclass will provide at minimum the following information:

- a string explaining the response
- whether aborting the current activity/process is recommended
- a reference code (type resRefType) that the host will use to refer to the anomaly
- the expectation group id for which the primary expectation violation occurred.

The actual response class can also be used to determine what should be done. In particular, the mclMonitorCorrectiveResponse class indicates that MCL is recommending corrective action be taken to address an anomaly. This class provides the following additional function:

- `pkType responseCode();`

The value of this type may be checked against the symbols in mclSymbols.h (see the Symbols document for more information) in a switch statement to decide what should be done in the host to address the corrective response.
Once the host has decided what to do with each individual response, it should inform MCL of what action has been taken. The primary API functions for updating MCL on the state of a recommendation are:

- `suggestionImplemented(string key, resRefType referent);`
- `suggestionFailed(string key, resRefType referent);`
- `suggestionIgnored(string key, resRefType referent);`
- `provideFeedback(string key, bool feedback, resRefType referent);`

The following code is adapted from the Mars Rover's UGPA controller:

```
void generic_controller::handle(responseVector mcl_rv) {
  for (responseVector::iterator rvi = mcl_rv.begin(); … ; rvi++) {
    if ((*rvi).recommendAbort()) abort();
    if ((*rvi).requiresAction()) {
      mclMonitorCorrectiveResponse *mcr = dynamic_cast<...>(*rvi);
      if (mcr) {
        switch(mcr->responseCode())
        case CRC_ACTIVATE_LEARNING:
          activate_learning();
          suggestionImplemented("rover",mcr->referenceCode());
          break;
        otherwise:
          suggestionIgnored("rover",mcr->referenceCode());
          break;
      }
    }
  }
  // <dispose of response vector>
  …
}
```

Note how the response vector is processed and MCL is apprised of the actions taken at the host level.

Less Intrusive MCL Development Using Wrappers

Typical integrated deployment requires somewhat extensive changes to the host system. A less intrusive deployment model takes advantage of object-oriented design by "wrapping" major host control systems in objects whose only purpose is to handle MCL specifics. This technique will not be appropriate for all host types, but for those situations where it applies, the underlying host system's control code will remain relatively pristine after the integration with MCL is complete.

We will use the MonCon monitor & control simulation substrate as a running example for the wrapper-based MCL deployment discussion. MonCon implements classes for domains, agents, controllers, and actions. Using this model, each agent in the simulation will have its own MCL key for independent anomaly handling. We will further seat the discussion in the Mars domain, using simulated Mars Rovers, which are all implemented on top of the MonCon substrate.

The basic idea with wrappers is to take a single class that performs multiple functions, such as the MonCon class “agent”:

```
class agent {
  agent();
  virtual initialize();
  …
};
```

and generate a “wrapper class” that selectively implements virtual classes of the original class or passes them through to an internal object of the original class:

```
class agent_wrapper : public agent {
  agent_wrapper(agent* a) : agent(),inner_agent(a) {};
  virtual agent* peel() { return inner_agent; };
  virtual initialize() { return peel()->initialize(); };
  agent* inner_agent;
}
```

The Mars Rover MCL integration is accomplished using MCL wrappers for the agent, controller, and action classes. The agent class implements the initialization phase of the MCL deployment in its own initialize() method. The controller class implements monitoring and anomaly handling by sequencing monitor() calls and using the original controller's API to implement MCL's recommendations. Finally, the action wrapper implements expectation group and individual expectation management for the individual actions.

Most relevant snippets of code are included as an example.

```
bool mclAware::mcl_agent::initialize()  {
  bool rv = declare_mcl();
  rv &= peel()->initialize();
  return rv;
}

bool mclAware::mcl_agent::declare_mcl() {
  if (settings::getSysPropertyBool("moncon.usemcl",true)) {
    string kee = mcl_key_of();
    // initialize with MA MCL API
    mclMA::initializeMCL(kee,0); // assumes synchronous
    // now specify configuration
    if (get_active_controller())
      mclMA::configureMCL(kee,domain_of()->class_name(),
                  peel()->class_name(),
                  get_core_controller(get_active_controller())->class_name());
    else
      mclMA::configureMCL(kee,domain_of()->class_name(),class_name());
    // now register observables (sensors)
    for (observable_list::iterator sli = observables()->begin();
       sli != observables()->end();
       sli++) {
      // declare the sensor the best you can
      mclMA::observables::declare_observable_self(kee,(*sli)->get_name());
      mclMA::observables::set_obs_prop_self(kee,(*sli)->get_name(),PROP_DT,
                                mc2mcl_odt((*sli)->get_odt()));
      mclMA::observables::set_obs_prop_self(kee,(*sli)->get_name(),PROP_SCLASS,
                                mc2mcl_osc((*sli)->get_osc()));
    }
    // new with the observables update...
    // now create the mcl observable updater
    mclMA::observables::update mcl_update_obj;
    fill_out_update(mcl_update_obj);
    mclMA::updateObservables(kee,mcl_update_obj);
    // and initialize it over on the MCL side
    return true;
  }
  else
    return false;
}
```

Controller level

```
bool mclAware::mcl_control_layer::control_out() {
  bool rv = peel()->control_out();
  // set up an update and monitor
  mclMA::observables::update mcl_update_obj;
  my_mcl_agent->fill_out_update(mcl_update_obj);
  responseVector mclrv = mclMA::monitor(mcl_key_of(),mcl_update_obj);
  // at this level, just tell MCL we are ignoring everything
  for (unsigned int k=0;k<mclrv.size();k++) {
    if (mclrv[k]) {
      mclMonitorResponse* tmr = mclrv[k];
      mclMA::suggestionIgnored(mcl_key_of(),tmr->referenceCode());
      delete tmr;
      mclrv[k]=NULL;
    }
  }
  return rv;
}
```

Action level

```
bool mclAware::mcl_action::activate() {
  // do the MCL stuff here...
  mclMA::declareExpectationGroup(mcl_key_of(),eg_key_of(),
                         get_parentref(),get_resref());
  // now declare expectations...
  // this is a virtual function that needs to be extended.
  declare_expectations();
  return peel()->activate();
}

bool mclAware::mcl_action::deactivate() {
  // do the MCL stuff here...
  bool rv = peel()->deactivate();
  unset_resref();
  unset_parent();
  mclMA::observables::update mcl_update_obj;
  my_mcl_agent->fill_out_update(mcl_update_obj);
  mclMA::expectationGroupComplete(mcl_key_of(),eg_key_of(),mcl_update_obj);
  return rv;
}
```

```
bool mmcl::rover_mcl_action::declare_expectations() {
  model* mm = rover_action_of()->ptr_to_my_model();
  if (!mm) {
    uLog::annotate(MONL_HOSTERR,"no model for "+describe());
  }
  action_model* mam = dynamic_cast<action_model*>(mm);
  if (mam) {
    for (unsigned int ai = 0; ai < mam->number_of_effects(); ai++) {
      action_effect* cae = mam->get_effect(ai);
      mclAware::expgen::ae2expectations(mcl_key_of(),eg_key_of(),
                              rover_action_of()->rover_of(),cae);
    }
  }
  else {
    uLog::annotate(MONL_HOSTERR,"no action model for "+describe());
  }
  return true;
}
```