README for Multilevel Coordinator CoABS Grid Component v1.0 beta
University of Michigan

Contact Brad Clement at bradc@umich.edu for questions, comments,
and suggestions.

Contents of this file:

I. Installation Instructions

The Multilevel Coordinator Grid component is currently configured to run
on
Intel-Linux machines. The software distribution is available over the web
at

http://ai.eecs.umich.edu/people/bradc/mapc/um-mlc-v1.0b.tar

It will untar into a directory named um-mlc-v1.0b. Inside are the
following
subdirectories and several data and configuration files:

Subdirectories

BBNGridProxy	- BBN's proxy code for accessing the Grid from C
c1	- C code for the coordinator, coordinator1, and factoryAgent
c2	- C code for coordinator2 and neoAgent
neo	- soft link to directory of plan files for NEO demo
plans	- used by coordinator for temporary IO of plan files

General files

README.txt	- this file
compile.sh	- script to compile all agents for the two demos
coord.sh	- script launching the multilevel coordinator
coord1.sh	- script launching coordinator1
coord2.sh	- script launching coordinator2
coordinator	- multilevel coordinator executable
coordinator1	- coordinator1 executable
coordinator2	- coordinator2 executable
factoryAgent	- coordination client executable used for factory demo
neoAgent	- coordination client executable used for NEO demo
setvars	- script setting path variables for other scripts

Files for Factory Demo
----

fm.sh	- script launching facility manager

fmplans	- facility manager's plans
im.sh	- script launching inventory manager
implans	- inventory manager's plans
init_state.plan	- initial state
pm.sh	- script launching production manager
pmplans	- production manager's plans
factory.eps	- encapsulated postscript file showing coordination results

Files for NEO Demo
----

agent1.info	- plan-file mapping for agent1
agent1.plans	- agent1's plans
agent1.sh	- script launching agent1
agent2.info	- plan-file mapping for agent2
agent2.plans	- agent2's plans
agent2.sh	- script launching agent2
env_agent.info	- plan-file mapping for the environment agent
env_agent.plans	- environment agent's plans
env_agent.sh	- script launching environment agent

Edit the setvars script and specify the paths to the local CoABS Grid installation and to the um-mlc-v1.0b directory where the above files were installed. Also, make sure the soft link to the neo directory exists.

For compiling or running any of the agents demonstrated, the LD_LIBRARY_PATH environment variable will likely need to point to Java Runtime shared object libraries which are used by BBN's proxy code. For example,

```
setenv LD_LIBRARY_PATH
/usr/local/jdk1.2.2/jre/lib/i386:/usr/local/jdk1.2.2/jre/lib/i386/classic:
/usr/local/jdk1.2.2/jre/lib/i386/native_threads
```

If the compiler has problems locating libjvm or scripts do not work because shared object libraries cannot be located, it is likely because this variable was not set.

If you need to recompile the agent executables, first edit the Makefiles in the c1 and c2 directories to set the appropriate "include" directories for JDK and the JRE shared library directory. The compile.sh shell script in the top directory compiles all of the C code in each of the c1, c2, and BBNGridProxy directories and moves the executables to the top directory where they are used by scripts for running two demos as described next.

II. Running the Demos

The Multilevel Coordinator can be launched once and service the client agents for both of these examples. First run the coord1.sh and coord2.sh shell scripts to launch two coordinator slaves. Once these have registered to the grid, run the coord.sh script to start the master coordinator that serves as an interface between the other two and clients requesting coordination and plan summarization. These three Grid components together comprise the Multilevel Coordinator. Once these are launched, scripts for the client agents can be started.

The Factory Demo involves three managers in a factory that coordinate their activites using the Multilevel Coordinator. Three agents representing these managers first request that the coordinator * summarize* plan information for their hierarchies and then request to have their plans *coordinated* with others. Detailed descriptions of the messages and output for these agents are given in Section III. Start up the factory agents with the fm.sh, pm.sh, and im.sh scripts in separate terminals to better follow the output.

The NEO demo shows how agents can use the Multilevel Coordinator to coordinate their plans to avoid conflicts while executing and reacting to change in the environment. Here, two transport travel in a network of locations to pick up evacuees, bring them to safety points, and destroy routes to prevent danger from following from behind. This is also described in more detail later, but to run it, start the env_agent.sh, agent1.sh, and agent2.sh scripts.

III. Explanation of Demos

Factory Demo

A production manager (the pm agent) is responsible for creating a CD part using three machines (M1, M2, and M3) that consume parts to produce others as follows:

Machine	Parts consumed	Parts produced
M1	A, DUMMY_A	A_PRIME
	C_PRIME, D	CD
M2	A_PRIME, B	AB
	C, DUMMY_C	C_PRIME
M3	C_PRIME, D	CD

The production manager has a hierarchical plan for building the CD part using M1 to produce A_PRIME, M2 to produce AB, M2 to produce C_PRIME, and a choice of either M1 or M3 to produce CD. The plan specific hierarchies for each of the agents are detailed in the pmplans, implans, and fmplans files, and the initial state is given in the init_state.plan file and is as follows:

```
(available "A");
 (available "B");
 (available "C");
 (available "D");
(not (available "E"));
(not (available "F"));
```

```
                   (free "M1");
                   (free "M2");
                   (free "M3");
                   (not (holding "M1" "TOOL"));
                   (not (holding "M2" "TOOL"));
                   (not (holding "M3" "TOOL"));
                   (available "DUMMY_A");
                   (available "DUMMY_C");
                   (not (available "AB"));
                   (not (available "CD"));
```

The parts must be "available" on the space limited shop floor in order to use the machines to produce them. The inventory manager (im) has goals to make E anf F available, but there are restrictions on which parts can occupy space on the floor at the same time. The inventory manager's hierarchical plan involves either picking up E to make it available, swapping E in for AB, or swapping E in for A. It also gets F on the floor by either picking up F, swapping F for CD, or swapping F for C. For each of the two set of parts, only one can occupy space on the floor at a time:

(A, A_PRIME, AB, E), (C, C_PRIME, CD, F)

In addition the facility manager must service each machine once by equipping it with a tool and then performing maintenance. The machines are unavailable for production while they are being serviced. The facility manager's hierarchical plan branches into choices of servicing the machines in different orders.

The coordination problem involves finding synchronizations of the agents' tasks (possibly at abstract levels using summary information derived for the plans) such that they all achieve their goals within the shortest time (collectively). Different coordinated plans found by the coordinator are depicted in factory.eps in the top directory. The optimal one at the bottom is the one that the coordinator returns to the agents.

Details about the communication protocol among the factory and coordinator agents is detailed in Section IV. Basically, the agents send the coordinator their plan hierarchies and request summary information to be derived for the abstract plans. Then they request coordination for their summarized plans. The coordinator returns synchronization and block restrictions on their plans. Then, the agents execute their plans while waiting for and signaling each other according to the temporal restrictions given by the coordinator. In order for the agents to know which particular instantiations of plan operators from certain agents they should wait and signal on, plans are referred to by strings of the format "<agentname>.<plan goal>.<id>" where the id is an index of the instantiated plan in the hierarchy as it is level-order expanded from the top down. These strings are given to the agents in their summarized plans. Because the NEO demo uses different coordination service types and protocols, the message formats and output are fairly different.

NEO Demo

In this demo, there are two agents AGENT1 and AGENT2 who are tasked with transporting evacuees from unsafe nodes to certain safe nodes. AGENT1, initially at node 1, has to transport evacuees from nodes 2, 6, and 5 (unsafe nodes) to node 1 (safe node). AGENT2 initially at node 7, has to transport evacuees from nodes 4, 3, and 8 (unsafe nodes), to node 7 (safe node). The edges between the nodes represent links that the agents can use to move between nodes. AGENT1 is tasked with destroying links (2,6) and (6,5), while AGENT2 is tasked with destroying link (3,6). Only one agent can traverse a link at a given time, but nodes are large enough to hold both AGENT1 and AGENT2. AGENT1 and AGENT2 are capable of two kinds of primitive actions. The first kind is called MOV and it is used to transport evacuees between two nodes. MOV(Q,X,Y) is used to denote the transport of evacuees by agent Q from node X to node Y along the link connecting them. The other action called MOVD is used to effect a MOV and then destroy the link that was traversed.

For the topology of the transport network and a graphical representation of the hierarchical plans of the agents see the following page:

http://ai.eecs.umich.edu/people/bradc/mapc/sim/Evac.html

There are three different kinds of messages that are displayed during a run:

1. Plan operator execution message: This message is used to signal the completion of the execution of a primitive operator. The format of this message is:

```
Link traversal completed/Link traversal and destruction completed.
< X Start_Node End_Node >
```

which denotes that agent X has moved from Start_Node to End_Node (sometimes destroying the link after traversing it).

2. Agent synchronization message: This message is used to denote the transmission of a synchronization message (signal) between two agents. The format of this message is:

```
X has sent a signal (Y) to Z  or
X has received a signal (Y) from Z
```

where X and Y are agents and Y is the signal identifier (an integer). An agent sends a signal to another agent to notify the latter of the completion of an action which has temporal precedence (decided by the coordinator) over another action to be executed by the latter agent. For example, if AGENT2 uses link (2,6) to evacuate from node 8, AGENT1 must refrain from destroying (2,6) until after AGENT2 has finished using that link. Therefore, AGENT2 sends a signal to AGENT1 after it has crossed link (2,6) on the way back to its safe node 7. After receiving this signal, AGENT1 can cross link (2,6) and then destroy it.

3. Operator conflict resolution message: This message is used to show how potential conflicts between plan operators (both at abstract and primitive levels in the plan hierarchies) are resolved by appropriate temporal ordering of the operators. We use temporal relations from Allen's interval algebra to order plan operators. The format of this message is:

```
(X Operator <P>) R (Y Operator <Q>)
```

where X and Y are agents executing operators P and Q respectively. Operators P and Q are temporally ordered by the relation R.

IV. Multilevel Coordinator Interface

The coordinator currently has two basic services:

1) summarize plans
   The coordinator builds summary conditions for abstract plans from the plans in their respective decompositions. Given a library of plans and a top-level plan/goal, the coordinator will instantiate the hierarchy, summarize conditions for each instantiated plan in the hierarchy, and return the hierarchy. The format of the message for requesting summarization is

   ```
   "summarize <top-level-goal> <plan list>"
   ```

   The plan list is a string of concatenated plans of the format described later in this section.

2) coordinate plans

The coordinator collects plans from agents requesting coordination and coordinates them in batches based on a specified cycle time. Agents can repeatedly ask for coordination as they execute their plans and add new goals. The coordinator assumes that an agent's plan hierarchy is internally consistent--meaning that no action will cause another action to fail if a top-level plan is decomposed and executed in the absence of other agents (or other unforeseen events). The coordination has three optional modes for each batch:

1) summarized - The plan input is already summarized in order to enable abstract reasoning.

2) not_summarized - The plan input is not summarized. In this case, the coordinator summarizes and coordinates the plans

   For both of the above modes, plan modifications are sent back in messages to the appropriate agents at which point they can execute free of conflict by obeying the modification constraints. The format of the plan modification message includes a list of point constraints over the endpoints of pairs of plan executions followed

```
  by a list of blocked plan choices that cannot be selected during
  execution.

  modifications
  <plan1> start|end <plan2> start|end P|PS|SF|F
  <plan1> start|end <plan2> start|end P|PS|SF|F
  . . .
  blocked_plans
  <plan1> <plan2> . . .

  Plans are referenced with a string of the form
  "<agentname>.<plan goal>.<id>" as described in the factory demo
  description.  The first "start" or "end" refers to an interval
  endpoint of plan1, and the second "start" or "end" refers to an
  endpoint of plan2.  P means the first endpoint must Precede the
  second.  F = Follows and S = same time.  PS means precedes or same,
  etc.

3)  during_execution - Agents are coordinated while they execute their
                      plan hierarchies.  Agents send elaborations
                      to the coordinator while the coordinator informs
                      the agents of plans safe to execute and safe
                      synchronizations of their abstract or primitive
                      plans.

Agents request coordination with a message of the following format:

coordinate <mode> <top-level-plan> <planlist>

See http://ai.eecs.umich.edu/people/bradc/mapc/mapc.html for more
information about multilevel coordination and animated simulations
of coordination in NEO domains.

Below are the command line arguments necessary to launch the different
coordinator and client executables.

coordinator <BATCH_CYCLE_SECS> <COABS_PATH> <INSTALL_PATH>
   This is the Multilevel Coordinator executable that interfaces
   to all client agents.

   BATCH_CYCLE_SECS is the number of seconds after the first agent
   requests coordination that it collects other agents requests before
   coordination begins.  All other requests are saved until the
   coordinator completes the session.

   COABS_PATH is the path to the CoABS Grid installation.

   INSTALL_PATH is the path to the installation of this Multilevel
   Coordinator component.

coordinator1 <INIT_STATE_FILE> <COABS_PATH> <INSTALL_PATH>
   This component registers separately on the grid but handles
   summarization and uses summary information to coordinate
   agents prior to execution.
```

   INIT_STATE_FILE is the file that contains the current state of
   the world. It is in the format of a plan with the state listed
   as postconditions. See the plan format at the end of this section
   and the example init_state.plan in the top directory.

coordinator2 <log_file> <COABS_PATH> <INSTALL_PATH>
   This component registers separately on the grid but performs
   coordination during execution.

   log_file - logged output of all coordination events

factoryAgent <agentname> <COABS_PATH> <INSTALL_PATH>
   This is an example agent that requests coordination using summary
   information and executes it's plan based on modifications suggested
   by the cooordinator. This is used to start up the factory agents.

neoAgent <log_file> <info_file> <plan_file> <COABS_PATH> <INSTALL_PATH>
   This is an agent that executes its plan while being coordinated.

   log_file is the event log file.

   info_file contains agent plan information mapping goals to files.

   plan_file contains the task agent's plans

An agent plan is composed of the following sections:

1. NAME: This is a string identifier for the plan which is used for
   labeling purposes.

2. GOAL: This section specifies the goal that successful execution of
   plan will satisfy. It consists of a string identifier followed by an
   optional list of parameters.

3. PRECONDITIONS: This section specifies the conditions that must
   hold immediately before a plan can be executed. It is specified by a
';'
   delimited list of predicates.

4. INCONDITIONS: This section specifies the conditions that must
   hold while a plan is being executed. It is specified by a ';'
   delimited list of predicates.

5. POSTCONDITIONS: This section specifies the conditions that must hold
   immediately after a plan has been executed successfully. It is
specified
   by a ';' delimited list of predicates.

6. DURATION: This optional section specifies the duration of execution of
    the
   plan.

7. BODY: The body of a plan describes a sequence of actions to be taken in order to accomplish a goal. The actions might be of the form

```
ACHIEVE SUBGOAL [SUBGOAL ARGUMENTS]
```

which results in the execution of a subplan which has SUBGOAL as its goal, or

```
EXECUTE PRIMITIVE-ACTION [PRIMITIVE ACTION ARGUMENTS]
```

where PRIMITIVE-ACTION is a directly executable action.

Further, ACHIEVE and EXECUTE statements may be combined by the keywords AND and OR as in the following action statements:

```
AND {
   ACHIEVE SUBGOAL-1 [SUBGOAL-1 ARGUMENTS]
   .
   .
   .
   ACHIEVE SUBGOAL-N [SUBGOAL-N ARGUMENTS]
   }
```

The semantics of this statement is that subgoals SUBGOAL-1 through SUBGOAL-N must be achieved in the stated order for the composite action to succeed.

```
OR {
   ACHIEVE SUBGOAL-1 [SUBGOAL-1 ARGUMENTS]
   .
   .
   .
   ACHIEVE SUBGOAL-N [SUBGOAL-N ARGUMENTS]
   }
```

The semantics of this statement is that any one of SUBGOAL-1 through SUBGOAL-N can be achieved in order for the composite action to succeed.

A simple plan file follows:

```
PLAN {

   NAME : STACK

   GOAL : ACHIEVE STACK $X $Y;

   PRECONDITIONS : (CLEAR  $Y);

   INCONDITIONS : (NOT (CLEAR $Y));

   POSTCONDITIONS : (NOT (CLEAR $Y));
                           (ON $X $Y);

   BODY :

               EXECUTE STACK-OPERATOR $X $Y;
}
```