MIDCA SOTU (State of The Undertaking)

[Cox: What about Dustin’s sandbox?]

[Where is other documentation?]

About this document: This document is intended to

A) provide a picture of the current state of the MIDCA architecture (version 1.2), and

B) suggest a series of potential tasks that could be undertaken to improve and build upon it, with time estimates for each.

It is divided into sections based on a logical division of the functionality of MIDCA. Each section attempts to list the files which make up the discussed functionality, but such lists may be incomplete. Possible tasks are presented with the parts of the architecture they relate to, and a complete task list is presented at the end of the document.

Section I: Core Architecture

1. Phases and Modules [phasemanager.py]

The MIDCA architecture runs by repeatedly cycling through a number of phases. Each phase corresponds roughly to a box in the MIDCA diagram [citation needed]. For example there is an “Eval” phase, and a “Plan” phase.

Each phase that is to do anything must have an associate module. A module is a class in python which follows a certain pattern (i.e. methods init() and run() are required with specified parameters).

In the current implementation, phases and modules are completely customizable. A user can specify any named phases in any order and assign arbitrary classes (within the restrictions mentioned above) to implement them. The only mandatory phase is simulation; this is when world events either happen (in simulation) or are reported to MIDCA (in a real-world or real-time environment).

**High priority. Task 1[~1 week]: Clean up module extension procedures. This would involve adding a base class for MIDCA modules, minor improvements to error handling, and simplifying files structures. It would not really change the process of extending MIDCA, but it should make it a bit easier to understand.**

1. Cycle [phasemanager.py; midca.py]

The MIDCA cycle begins by initializing all modules. If a module has an init method, it is initialized with a ‘world’ argument representing the start state, and 2 other arguments mem and memkeys. See the memory section for a discussion of these last two args. This method is called in execution order for each module. A warning message is given for modules that do not have init methods.

Once all modules are successfully initialized or skipped, the cycle starts. In a user-given order, each module’s run() method is called. The run method takes only one argument, ‘verbose’, for debugging. There is a simple text-based control interface that allows MIDCA to run anywhere from one phase to many cycles and prints output to the terminal.

**High priority. Task 2 [~1 weeks]: Do real logging – have MIDCA create a log file for each run and send output there. Organize log files in a useful structure.**

1. Customization [custom/customsetup.py; custom/option/options.py]

As mentioned above, the MIDCA cycle is fully customizable. Customization happens in the customsetup file. There is a variable PHASES which controls the names and ordering of phases. There are also two mandatory generator methods that populate the phases with the modules which implement their functionality. The first method is get\_world\_sim(), which returns a valid MIDCA module which either simulates the world or updates MIDCA’s view of the world. This will be run at the beginning of every cycle, including the first. The second method is get\_modules(), which returns a dictionary (python implementation of a hash map) in which the keys are phase names and the values are valid MIDCA modules. For a module to be run in the MIDCA cycle, its name must be included in the PHASES list and that same name must be in the get\_modules() dictionary.

The second layer of customization is options. This functionality allows MIDCA to be run differently over different runs without changing the underlying code. Essentially an option sets a kind of global variable in MIDCA which modules can check either at initialization or at runtime. A user can set which options are available in the options.py file. When MIDCA is loaded, it will ask for the name of a valid options file. If none is given, there is a text-based interface to set the values of all options, and the configuration can be saved to a file if desired. In the current implementation, different options control whether there will be an arsonist, how long it takes to catch him if so, how goals are prioritized, etc.

**Low priority. Task 3 [2-3 weeks]: Create a GUI for generating a MIDCA cycle. This would involved tools to create phases and move them around, as well as the ability to assign implementing modules.**

**Low priority. Task 4 [~1 week]: Create a GUI for options. This would give the user the ability to add, remove and modify options, and to more easily save and load configurations. However, actually using option values in code would still have to happen outside the GUI.**

1. Memory [mem.py]

All communication between modules/phases in MIDCA takes place through memory. Right now there are no real restrictions on what memory is or how it is used. Any data structure (e.g. list, dictionary, object, etc.) can be assigned to a key in the central memory in any module, and accessed by that keyword in any other module. The syntax for this is a bit convoluted and should probably be streamlined; this would be a piece of task 6 below.

The paradigm itself is powerful but mistake-prone; it means that for two modules to work together they must each know not only what type of data is going to be passed but also what keyword it will be associated with. As has been discussed in the MCL group, it would be valuable to have a single language for memory storage, but so far we have no real candidates.

**Med priority. Task 5 [1-2 weeks]: Have MIDCA generate a graphical representation of how memory is used. Specifically, it would show which modules access which memory locations for retrieval and storage, and create a dependency graph for modules (which might look something like the MIDCA diagram). This could be helpful for showing people exactly how MIDCA works and would also be a useful development tool.**

**Unknown priority. Task 6 [~1 week]: make some minor changes to streamline memory usage. Use constants rather than strings as memory indices (also attributes rather than keys; this is technical and python-specific). Create a no-arg memory.get\_previous() method which returns the last value inputted in the phase before the current one; this would allow, for example, the ‘assess’ phase to quickly and intuitively retrieve the last result from the ‘note’ phase, assuming assess immediately followed note. This is in keeping with the theoretical idea of MIDCA information flow.**

Section II: Module Functionality

This section describes the implementations of modules that are currently used in MIDCA. As described above, modules are intended to be fully customizable and to be swapped in and out, so the functionality described in this section is not considered to be a settled state. Many modules are very specifically tailored to certain domains; this is mentioned where applicable.

1. Simulation and Planning [worldsim/\*; modules/plan/\*; modules/simulator.py]

The current world simulator is a relatively simple engine that models a world using a predicate representation. It is capable of reading domain representations and world states from file, using a language both substantially less powerful and simpler than PDDL. It can simulate actions in the world, and has several convenience methods to answer queries about the current state and which actions are legal. As such this simulator can easily be used in most planning algorithms. However, as mentioned the domain scripting language lacks some expressiveness in terms of the types of states in can represent.

The current planner is a python version of the SHOP hierarchical task network planner. There is also an available module that uses the more complete SHOP2. However, SHOP2 is written in LISP, so due to unavoidable translation/communication issues and a more avoidable inefficient implementation, that version runs much slower even on very simple problems. The planning module

**Low priority. Task 7a [1-4 weeks]: extend the simulator to accept PDDL domain files. The large range is due to the fact that I don’t know PDDL that well and am not sure exactly how large the changes will need to be.**

**Unknown priority. Task 7b [1-2 weeks]: Extend the python version of shop to incorporate some of the features of SHOP2 (especially the ability to interleave methods from different tasks and ways to optimize search). Integrate this and an A\* algorithm to give MIDCA two alternate ways to approach planning problems. Users would need to provide heuristics for A\* and methods for HTN, but could do so in such a way that MIDCA could access and change them if desired.**

**Med priority. Task 8 [1-2 months]: Fully integrate SHOP2 with MIDCA. This would include using SHOP2 as a simulation engine, and giving MIDCA access to SHOP2 methods. It would also mean running a LISP interpreter alongside the python one and communicating back and forth.**

**Note: only one of 7a/b or 8 should be undertaken, since they are redundant. Also, it would be wise to contact Vikas Shivashankar and/or Dana Nau for help with either project.**

1. Domain-specific Modules [all are in modules/: evaluate/evaluate.py; goalinsertion/\*]

Goal insertion: This module generates new goals for the blocksworld/fire domain based on several domain-specific processes. These processes could be extended to new domains, but all would require significant user effort for each new domain added. The specific processes used are: a) loading user-defined goals; b) using predefined Tilde-Foil trees to generate goals, c) using Meta-AQUA to generate arsonist-catching goals. Which of these methods is/are active is defined by options as described in 1.3 above.

**Med priority. Task 9 [<1 week]: Create a standard for goal representation that can be understood both by MIDCA and by whatever planner we settle on. Users can then generate goals using this representation.**

Evaluate: The evaluate module checks to see if the current goals have been accomplished. It is domain-specific because it is intended to provide a scoring function for how the agent is doing in a block-stacking world where fires occur, and the method is specific to finding this information.

**High priority. Task 10 [< 1 week]: write a simple Eval module which checks if all goals for the current goalset have been achieved, sets the current goalset to empty if so, and records the Boolean result of this operation in memory. Users could then extend this module with any other functionality they want.**

1. Domain-general modules [utils/adist; utils/gng.py; from modules/: observe/\*, note/\*, intend/\*, execute/\*, assess/\*]

The modules in this section are domain-general, but many of them still rely on having a particular representation of the world. Specifically, they use classes from the world simulator described above which is based on a predicate state representation.

Observe: Simple copies the previous world state

Note: Uses the A-distance on a predicate representation of the world state to determine if an anomaly has occurred. Also provides statistical information on the nature of the anomaly. While domain-general, this method may require tuning to work well in different domains.

Assess: Uses information from note to classify anomalies into one of several dynamically generated statistical clusters. While technically domain-general, the effectiveness of this method has not been tested on a wide variety of domains and it may require tuning.

Intend: Chooses a goal set to plan for based on assigned goal priorities. The priority of a set is the max of the priorities of the set’s members. Relies on goal generation algorithms providing goal priorities.

Execute: Loads the plan corresponding to the highest-priority goal set and executes the next action in that plan, if applicable.

**High priority. Task 11 [1-2 weeks]: General cleanup of these methods. Simplify them and make fewer assumptions about the nature of inputs and outputs.**

**Low priority. Task 12 [1-2 weeks]: Come up with a better structure for goals and plans. Allow for partial orderings of goals, more complex priority/importance assignments, and a greater ability for MIDCA to either maintain multiple plans and work towards each of them as appropriate or replan to incorporate new goals.**

Section III: Task List:

The following list contains all the tasks mentioned above. It is ordered purely by where in the MIDCA document the task came up. Note that there are more tasks here than I can reasonably do, but I hope this document will be a useful resource for people who work on MIDCA in the future. I will update it as I undertake and compete tasks from the list.

**Task 1 [~1 week]: Clean up module extension procedures. This would involve adding a base class for MIDCA modules, minor improvements to error handling, and simplifying files structures. It would not really change the process of extending MIDCA, but it should make it a bit easier to understand.**

**Task 2 [~1 weeks]: Do real logging – have MIDCA create a log file for each run and send output there. Organize log files in a useful structure.**

**Task 3 [2-3 weeks]: Create a GUI for generating a MIDCA cycle. This would involved tools to create phases and move them around, as well as the ability to assign implementing modules.**

**Task 4 [~1 week]: Create a GUI for options. This would give the user the ability to add, remove and modify options, and to more easily save and load configurations. However, actually using option values in code would still have to happen outside the GUI.**

**Task 5 [1-2 weeks]: Have MIDCA generate a graphical representation of how memory is used. Specifically, it would show which modules access which memory locations for retrieval and storage, and create a dependency graph for modules (which might look something like the MIDCA diagram). This could be helpful for showing people exactly how MIDCA works and would also be a useful development tool.**

**Task 6 [~1 week]: make some minor changes to streamline memory usage. Use constants rather than strings as memory indices (also attributes rather than keys; this is technical and python-specific). Create a no-arg memory.get\_previous() method which returns the last value inputted in the phase before the current one; this would allow, for example, the ‘assess’ phase to quickly and intuitively retrieve the last result from the ‘note’ phase, assuming assess immediately followed note. This is in keeping with the theoretical idea of MIDCA information flow.**

**Task 7a [1-4 weeks]: extend the simulator to accept PDDL domain files. The large range is due to the fact that I don’t know PDDL that well and am not sure exactly how large the changes will need to be.**

**Task 7b [1-2 weeks]: Extend the python version of shop to incorporate some of the features of SHOP2 (especially the ability to interleave methods from different tasks and ways to optimize search). Integrate this and an A\* algorithm to give MIDCA two alternate ways to approach planning problems. Users would need to provide heuristics for A\* and methods for HTN, but could do so in such a way that MIDCA could access and change them if desired.**

**Task 8 [1-2 months]: Fully integrate SHOP2 with MIDCA. This would include using SHOP2 as a simulation engine, and giving MIDCA access to SHOP2 methods. It would also mean running a LISP interpreter alongside the python one and communicating back and forth.**

**Note: only one of 7a/b or 8 should be undertaken, since they are redundant. Also, it would be wise to contact Vikas Shivashankar and/or Dana Nau for help with either project.**

**Task 9 [<1 week]: Create a standard for goal representation that can be understood both by MIDCA and by whatever planner we settle on. Users can then generate goals using this representation.**

**Task 10 [< 1 week]: write a simple Eval module which checks if all goals for the current goalset have been achieved, sets the current goalset to empty if so, and records the Boolean result of this operation in memory. Users could then extend this module with any other functionality they want.**

**Task 11 [1-2 weeks]: General cleanup of these methods. Simplify them and make fewer assumptions about the nature of inputs and outputs.**

**Task 12 [1-2 weeks]: Come up with a better structure for goals and plans. Allow for partial orderings of goals, more complex priority/importance assignments, and a greater ability for MIDCA to either maintain multiple plans and work towards each of them as appropriate or replan to incorporate new goals.**

**High priority. Task 13. Develop an interface with Lisp for Meta-AQUA (and versions of SHOP). Interact with Cox.**

**High priority. Task 14. Develop an interface with ROS. Should interact with act (execute?), perceive (observe?), and I suppose simulate modules. I would like to have an extremely simple demo with a Baxter. We could coordinate with Jared.**

**High priority. Task 15. Develop an API interface between object and meta-levels in MIDCA. Create a simple meta-level shell for all phases. Object level should produce a trace of “behavior” for each object-level phase. In its simplest form this will just be the logging facility in Task 2, although we should also add (and separate) and log for the meta-level itself for debugging purposes. Control of object-level from meta-level is unclear, but it should include the ability to change, delete, and add goals. Perhaps it could also change the ordering of the phases.**

**Med priority. Task 16. Incorporate Dustin’s Sandbox code into world simulator and/or logging code. The sandbox function is to enable experimentation and replication. It would need to interact with debugging/logging code, scoring functions in eval module, and code that determines resources and “problems” (i.e., arsonist). Optionally, I would like to dispel the notion of an initial state. Conceptually, an agent should always be “running.” Instead we would have the notion of a time machine that could go back into time and start again at some specific point. We should discuss the feasibility of the task further.**