Lab #4
Introduction to MOOS Programming

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Contents

1 Overview and Objectives

In today's lab we will produce our own MOOS applications. We begin by downloading an example application complete with it's own build structure. We proceed by making our first simple MOOS app emphasizing the usage of the basic MOOS components. This is followed by a couple more complex applications assignments.

1.1 Preliminaries

This lab does assume that you have a working MOOS-IvP tree checked out and installed on your computer. To verify this make sure that the following executables are built and findable in your shell path:

```
$ which MOOSDB
/Users/you/moos-ivp/MOOS/MOOSBin/MOOSDB
$ which uTimerScript
/Users/you/moos-ivp/bin/uTimerScript
$ which mykill
/Users/you/moos-ivp/scripts/mykill
```

1.2 MOOS, MOOS-IvP and Your Applications

In the previous lab we discussed the relationship between MOOS and MOOS-IvP. The MOOS tree is a body of software distributed as part of the MOOS-IvP tree as depicted in Figure 1. MOOS-IvP provides additional MOOS applications, including the IvP Helm behavior-based architecture, and has C++ build dependencies on the MOOS libraries. Today the focus is on building additional MOOS applications. Your MOOS apps will have a build dependency on the MOOS libraries, and you may choose to utilize libraries in the MOOS-IvP tree. We start by downloading the moos-ivp-extend tree which may be regarded as a template for extending the MOOS-IvP tree with apps and behaviors. This tree also includes a functional build structure to ease the learning curve on C++ build issues for now.

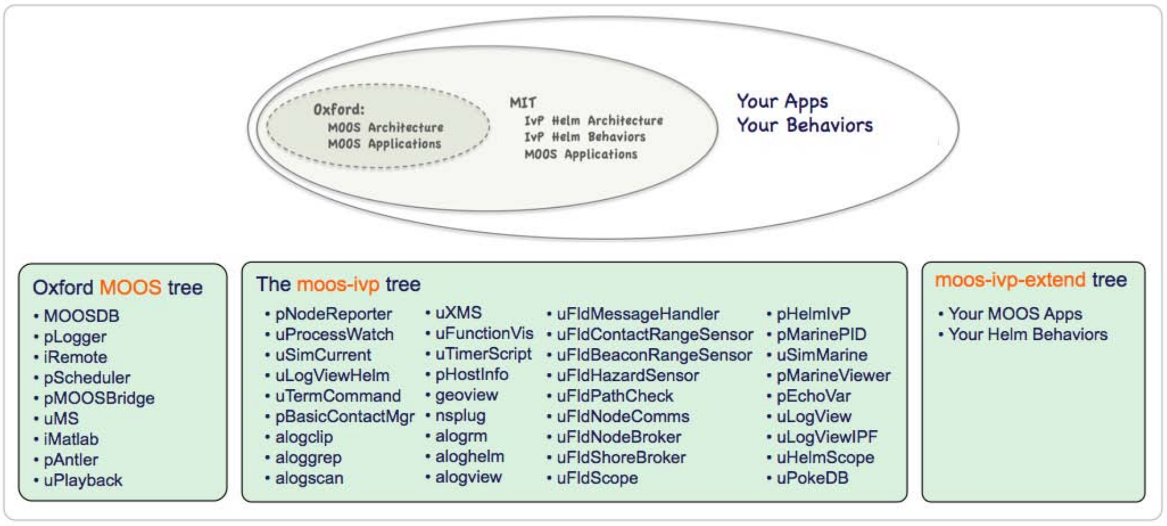

Figure 1: **Nested Repositories:** The MOOS-IvP tree contains the Oxford MOOS tree and additional modules from MIT including the Helm architecture, Helm behaviors and further MOOS applications. The set of modules may be expanded with user third-party applications or behaviors.

1.3 More MOOS / MOOS-IvP Resources

We will only just touch the MOOS basics today. A few further resources are worth mentioning for following up this lab with your own exploration.

- See the slides from the today's class which give a bit more background into MOOS and MOOS-IvP related to marine robotics.

- The Programming with MOOS documentation.

- The moos-ivp.org website documentation.
  http://www.moos-ivp.org

- MOOS Doxygen pages:
  http://gobysoft.org/doc/moos/class_c_m_o_o_s_app.html
  http://gobysoft.org/doc/moos/class_c_process_config_reader.html
  http://gobysoft.org/doc/moos/_m_o_o_s_gen_lib_global_helper_8h.html

1.4 The MOOS Application Structure

The main idea explored today is the notion and structure of a MOOS application. We know from the last lab that MOOS apps publish, subscribe for, and handle mail passed from one application to another through the MOOSDB. In the last lab we worked with existing MOOS apps, only modifying their functionality through configuration options provided by the app writer. Even without modifying the code of existing MOOS apps there are many ways to configure a system for unique domains. However, the real power of MOOS comes from the fact that no application is sacred. If you don't like what it does (or doesn't) do, you are free copy *and rename it*, and modify the code to your satisfaction. Or you can just build your own application from scratch. This is the focus of today's lab.

The key components to keep in mind in today's lab are shown in Figure 2 below. All MOOS apps begin by being a subclass of the MOOSApp superclass defined in the Oxford MOOS library. The primary work of the app developer is in writing the three functions shown in the figure.

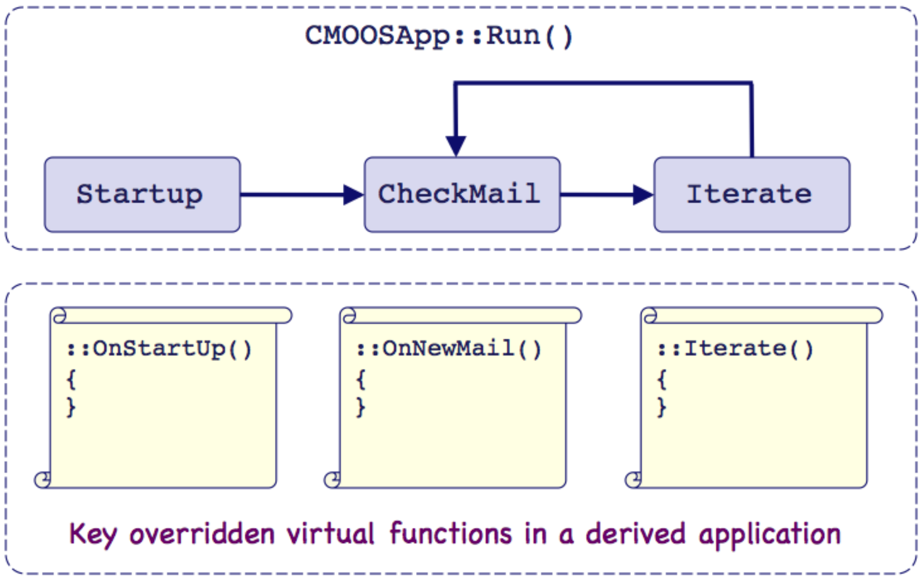

Figure 2: **The MOOSApp Key Functions:** All MOOS apps are a subclass of the MOOSApp superclass. Development mostly boils down to overriding the three functions below with the particulars of one's own liking.

1.5 Handy functions defined on MOOSMsg

The below methods are defined on an instance of MOOSMsg. In our pXRelayTest example app, we only call `msg.GetKey()` in Relayer.cpp, but all the below methods are available for getting the message fields. (See today's lecture notes for a description of a MOOS Message.) This information is probably in the MOOS documentation, but is repeated here for convenience.

```
// return the name of the message
std::string GetKey();
// return the name of the message (just another way)
std::string GetName();

// check data type is double
bool IsDouble();

// check data type is string
bool IsString();
```

```
// return time stamp of message
double GetTime();

// return double val of message
double GetDouble();

// return string value of message
std::string GetString();

// return the name of the process (as registered with the DB) which
posted this notification
std::string GetSource();

// return the name of the MOOS community in which the orginator lives
std::string GetCommunity();
```

2 Downloading and Building the moos-ivp-extend Tree

The first step of today's lab is to download the moos-ivp-extend tree. This tree may be regarded as a template for building your own set of MOOS applications or (later on) Helm behaviors. It contains a single MOOS app and a single Helm behavior, but more importantly a file structure and CMake C++ build system for build the source code. Eventually you will want to understand more about Make files and CMake files. But for now, by starting with a working template, the addition of new MOOS applications to the build structure is a simple matter of editing one or two files.

Your goals in this part are:

1. From a terminal window download a copy of the moos-ivp-extend tree with the following svn command:

   ```
   % svn co https://oceanai.mit.edu/svn/moos-ivp-extend/trunk moos-ivp-extend
   ```

   Verify that you can build this tree by:

   ```
   % cd moos-ivp-extend
   % ./build.sh
   ```

   It should add executable, pXRelayTest, in `moos-ivp-extend/bin/`. Add this directory to your path.

2. The `pXRelayTest` application built in this tree if very close to the `pXRelay` app distributed with the moos-ivp tree. Modify your xrelay.moos mission from Lab 3 (wget http://oceanai.mit.edu/2.S998/examples/xrelay.moos) to use the pXRelayTest application instead.

A Peek Under the Hood

Before moving on to the next exercise, take a quick look at the Relayer MOOS app class definition from this example. It is in Listing 1, and may also be found in `moos-ivp-extend/src/pXRelayTest/`. Note a couple things:

- The class declaration, on line 1, declares itself to be a subclass of CMOOSApp, the general superclass of all MOOS apps.
- Our class overrides the three key virtual functions of the parent class, declared in lines 7-9.
- Coding convention: All member functions declared before all member variables. Overridden MOOSApp member functions declared before subclass specific member functions.
- Coding convention: All member variables beging with `m_`, to set them apart from locally declared variables in the code.
- Coding convention: All member variables protected. Setting member variables happens through a public member function, e.g., lines 14,15.

Listing 1: The file `moos-ivp-extend/src/pXRelayTest/Relayer.h`*.*

```
#include "MOOSLib.h"
class Relayer : public CMOOSApp
{
 public:
  Relayer();
  virtual ~Relayer() {};

  bool OnStartUp();                          // Overriding key virtual function
  bool OnNewMail(MOOSMSG_LIST &NewMail);     // Overriding key virtual function
  bool Iterate();                            // Overriding key virtual function

  bool OnConnectToServer();
  void RegisterVariables();

  void setIncomingVar(std::string s) {m_incoming_var=s;};
  void setOutgoingVar(std::string s) {m_outgoing_var=s;};

 protected:
  unsigned long int m_tally_recd;
  unsigned long int m_tally_sent;
  unsigned long int m_iterations;

  std::string       m_incoming_var;
  std::string       m_outgoing_var;
  double            m_start_time_postings;
  double            m_start_time_iterations;
};
```

3 Building Your First MOOS Application

In this exercise we will build our first couple of simple MOOS applications. The function performed by the app is trivial: it handles a series of integers and determines if the product of the last N numbers is even or odd. But it also allows us to touch upon a few important basics of building a MOOS application.

3.1 Generating an Template App and Augmenting the Build System

Your goals in this part are:

1. Enter the moos-ivp-extend/src directory and run the GenMOOSApp script to build an application template

   ```
   % GenMOOSApp Sequence p ‘‘Jane Doe’’
   ```

   The above script, `GenMOOSApp`, should be in your path under moos-ivp/scripts/. If this is not in your path, add it. The above script invocation builds a new app called pSequence. The third argument is your name. You can type `GenMOOSApp -h` in the future to remind yourself.

2. Tip: Always put your name at the top of your source code files! This is a good practice in general, but even more so in a class setting, or an open source environment where people are sharing code.

3. Add your new app to the build system. Edit the file `moos-ivp-extend/src/CMakeLists.txt` and look for the line referring to `pXRelayTest`. Copy that as an example for your new app. Go back and re-run the build script. Check to see that pSequence has been built in the bin directory.

3.2 The Specs of Your New Application

Your application should meet the following specifications:

1. It should accept mail on an incoming variable containing an integer value.
2. The variable name may be configured by the user, with a reasonable default provided in your app.
3. After N mail messages have been received, your app should output a determination of whether the product of the last N numerical values is even or odd.
4. Your output should be in a string format and should be sufficiently descriptive of your result and the corresponding input, to allow another app to verify the result.
5. Your output variable name may be configured by the user, with a reasonable default provided in your app.
6. The number N is a parameter configurable by the user, with a default of 5.
7. You should handle the case of receiving a non-integer number by truncating.
8. Your app should be called `pSequence`.

General Hints: The key issues in this exercise are (a) how to handle configuration parameters in OnStartUp(), (b) how to handle incoming mail in OnNewMail(), and (c) how do some processing and posting of information in Iterate(). See the pXRelay or pXRelayTest applications for a good starting point.

3.3 A Second MOOS Application - to Test Your First App

Your next step is to build another MOOS app, pSequenceTester, to check the correctness of your first application. It should meet the following specs:

1. It should accept mail on an incoming variable containing the output of your pSequence application. Since the output variable used by pSequence is configurable by the user, the input variable for this app should be configurable too.
2. It parses the string input and determines whether the result concluded by pSequence is correct or not.
3. It publishes the results of the correctness check in another MOOS variable, the variable name of which is also configurable by the user.
4. Your app should be called `pSequenceTester`.

3.4 How to Test Your Applications

Thus far you have everything you need to test your work, except something to generate a sequence of random inputs to the pSequence application. To accomplish this, do the following:

1. Configure a uTimerScript script to endlessly generate a sequence of random integer postings, e.g., `NUM_VALUE=1999`, `NUM_VALUE=112`, etc.
2. Consult the uTimerScript documentation for the means of generating random values.
3. Make a single .moos mission file with the script and your two applications configured and launched with pAntler to demonstrate the full working setup.

4 Your Next MOOS Application - Prime Factorization

In this exercise we will build our next MOOS application. The function performed by the app is a bit harder: it handles a series of integers and determines, for each, the list of prime factors. A primary issue addressed in this exercise is how to build an app that performs an operation that may be considerably longer than the time slice allotted to a typical Iterate() loop. Your app should be able to handle the case where a simple query is preceded by a more complex query, without the simple query being blocked by the former. (Not unlike the 7-items-or-less aisle at the supermarket).

Getting Started

Begin by repeating the same steps from the previous example:

1. Enter the moos-ivp-extend/src directory and run the GenMOOSApp script to build the bare-bones application:

```
% GenMOOSApp PrimeFactor p ‘‘John Doe’’
```

2. Note: Please use the app name *pPrimeFactor*. It makes testing and grading much easier over many students.

3. Add your new app to the build system as before by editing the CMakeLists.txt and test that it builds.

4.1 The Specs of Your pPrimeFactor Application

Your application should meet the following specifications:

1. Your app should accept mail on an incoming variable containing an 64-bit integer value, max value of 18,446,744,073,709,551,616 (maximum for an unsigned long int). You can build your app under the assumption that the maximum query we will test for is 2^{48} however.
2. Note: The type `"unsigned long int"` is 64 bits long in most C++ implementations, but technically there is no guarantee. You may want to instead use the type `"uint64_t"` by including the `stdint.h` library.
3. The variable name may be configured by the user, with a reasonable default provided in your app.
4. Your app should determine the list of prime factors for a given input, and generate the following output for each:
   - an index indicating the order in which it was received,
   - an index indicating the order in which it was calculated,
   - the time it took to solve the factorization (use MOOSTime()),
   - the original number,
   - the list of prime factors.
   - a unique id specifying you, e.g., your Athena username.

5. Your output variable name may be configured by the user, with a reasonable default provided in your app.
6. Your application should not be blocked by large numbers. We should be able to see a response to low-number queries almost immediately, regardless of the size of the preceding query.
7. You should handle the case of receiving a non-integer number by truncating.
8. You are free to implement your app by building a cache of intermediate prime numbers. However, do not use a cache read in from a file, or generated and provided by another MOOS app. Each time your app starts, it should start from scratch in terms of cached knowledge.
9. Implement the –help, –interface, and –example command line switches discussed in the lecture.
10. Your app should be called `pPrimeFactor`.

General Hints: The key issues in this exercise are (a) how to perform your key operations in an Iterate() loop in an incremental manner to avoid blocking, (b) how to automatically tune your incremental operations so the Iterate() loop is also not sitting idle for too long.

4.2 A Second MOOS Application - to Test Your First App

Your next step is to build another MOOS app, pPrimeFactorTester, to check the correctness of your first application. It should meet the following specs:

1. It should accept mail on an incoming variable containing the output of your pPrimeFactor application. Since the output variable used by pPrimeFactor is configurable by the user, the input variable for this app should be configurable too.
2. It parses the string input and determines whether the result concluded by pPrimeFactor is correct or not.
3. It publishes the results of the correctness check in another MOOS variable, the variable name of which is also configurable by the user.
4. Your app should be called `pPrimeFactorTester`.

4.3 How to Test Your Applications

As with the previous exercise, you will need something to generate a sequence of random inputs to the pPrimeFactor application. In this case, you may want to game the inputs to test harder cases. To accomplish this, do the following:

1. Configure a uTimerScript script to endlessly generate a sequence of random integer postings, e.g., `NUM_VALUE=1999`, `NUM_VALUE=112`, etc.
2. Also generate a uTimerScript script for testing harder cases, e.g., non-blocking on large input numbers.
3. Make a single .moos mission file with the script and your two applications configured and launched with pAntler to demonstrate the full working setup.

5 Due Date and Grading Criteria

Due Date

All MOOS apps described in this lab are due during lab Thursday Feb 23rd. Apps will be tested during lab, and source code is due to the graders at the beginning of the lab.

Grading Criteria

Grading will be based on

1. Implementation Credit (80%)
2. Code Organization (20%)

Full implementation credit for each application will given based on whether or not you have met the given specs.

Code organization includes the issue of commenting your code, and choosing a robust code structure, e.g., properly initializing variables etc. Beginners to C++ programming can expect a healthy amount of constructive criticism in the first assignments.

Unfinished assignments will be tested and accepted in the next lab. A 5% deduction will be applied for each lab that goes by unfinished.