CLM Testing

Automated Tools for Testing CLM

Table of Contents

Why should you use Automated Tools to Test CLM?

By using automated testing tools you can automatically and easily run CLM under many different configurations and options, making sure that they still work with your changes. It also makes sure technical issues like restarts and change of processor count give correct answers. Doing this by hand is very time-consuming and difficult as well as requiring a lot of specialized knowledge about the code that is not necessarily obvious. Using the automated testing tools you can find problems with your code earlier in your development making issues easier to find. It will also make it easier for your code to be adopted in the CLM trunk.

Running the test suite is an expected part of the process for developments to move to the CLM trunk, and you are expected to run them before they will be adopted. The sooner you use them in your development process the easier it will be for you to find and fix issues. If you only use them at the end of your development process it may be very time consuming to find and fix issues at the end. And they may help you identify issues in your development and hence improve the robustness of your science.

What are the Different Testing Tools available?

There are two main test suites for CLM. One is for testing the code itself (the CESM Test Suite), and the other is for testing the CLM input file pre-processing tools such as mksurfdata_map (CLM Tools Testing). We address the CESM Test Suite here, and have a seperate page for the CLM Tools Testing with a link at the bottom of this page. There are also testing tools for the PTCLM python script, as well as a tool for testing the CLM build-namelist that we won't go into detail here. There are also unit testers for csm_share modules that we won't address either.

CESM Test Suite

The main workhorse for doing extensive testing of the software of CLM for many different resolutions and configurations is the CESM test Suite script "create_test" under the "scripts" directory. Our examples will all assume you are running on yellowstone and that you are using the "aux_clm_short" testlist which we recommend for quicker turnaround of your testing.

QuickStart to using create_test

Here we give a quick overview of how to use create_test. In the next section we will show more details of it's usage. Basic steps of usage are as follows. The first two steps are just deciding if you want to compare to a previous version (a baseline) and if you want to save the results of this test to compare future tests to.

1. Decide if you are going to compare to a baseline version of the code.
2. Decide if you want to save this version of the code as a baseline run.
3. If you are comparing to baseline you'll need to generate your baseline version by running the test suite on the baseline.
4. Run test suite on your version of the code.
    a. If you are NOT comparing to a baseline and do NOT want to generate a new baseline -- run without compare OR generate.
    b. If you are comparing to a baseline but do NOT want to generate a new baseline -- run with compare.
    c. If you are NOT comparing to a baseline but do want to generate a new baseline -- run with generate as in 3 above.
    d. If you are comparing to a baseline and your want to generate a new baseline -- run with compare AND generate.
5. Examine your test results
    a. Example test that PASS
    b. Example test that FAIL

Step 1 -- Decide if you are going to compare to a baseline version of the code.

The first thing to do is to decide if you want to compare the version of the code you are working on to a previous version. We call this creating a baseline version of the testing. The baseline version could be the given version of the code that you started from, or a previous version of the code with your changes intact that you tested and found to work well. If this IS TRUE -- you will do step-3 otherwise you will skip that step.

Why you wouldn't want to do this -- you already know your results are different anyway. You are just doing quick testing for functionality.

Why you WOULD want to do this -- to verify results are different when expected, and the same for cases where your changes should NOT effect simulations (for configurations they wouldn't affect for example with clm4_0 physics if you are only working on clm4_5).

Step 2 -- Decide if you want to save this version of the code as a baseline run for future testing.

The next thing to decide is if you want to save your test results for this round of testing to compare with in the future.If this is a significant step in your development, you WILL probably want to save results so you can compare future results to this case easily. If you are going to save it as a baseline, you will likely also want to save your code to your branch as well.

Why you wouldn't want to do this -- you didn't save this version on your branch so you can't go back to it easily. You are just doing quick testing for functionality.

Why you WOULD want to do this -- so you can compare future testing to this version.

Step 3 -- Generate the baseline version (If Step-1 is TRUE)

To generate new baselines use the "-generate" option. Generate creates baselines that can then be used later to compare other test results to. Then compare can be used later to compare to those baseline results. This can let you know if answers changed between two different runs of create_test. On yellowstone the default location to put the baselines for comparison is in "/glade/p/cesmdata/cseg/ccsm_baselines" which is only open to the group "cseg", so you'll need to specify a different directory to put your baselines in using the option "-baselineroot". So an example of using generate to create baselines is...

```
cd scripts
./create_test -xml_mach yellowstone -xml_compiler intel -mach yellowstone_intel -xml_category aux_clm \
-generate clm4_5_90 -baselineroot /glade/p/work/$USER/myclm_baselines
```

Step 4 -- Run the Test Suite on your version of the Code

You now run create_test for your code with options that depend on your answers to step 1 and step 2.

Step 4a -- Neither Step-1 NOR Step-2 TRUE -- run without generate OR compare

A sample of running it on yellowstone is...

```
cd scripts
./create_test -xml_mach yellowstone -xml_compiler intel -mach yellowstone_intel -xml_category aux_clm
```

This runs the CLM specific "aux_clm" testlist for the machine yellowstone with intel compiler on that machine and compiler. Note that you could run a testlist for a different machine/compiler to compare results between machines/compilers.

Results for the tests are put into the script "cs.status.<jobid> where <jobid> is the job ID number for that test submission. Tests that pass are clearly identified as "PASS" and there are various types of test failures that are reported as well.

Step 4b -- Step-1 TRUE, but not Step-2 -- run with compare

To then compare results with that previous version on a different branch or after you have made development modifications.

```
cd scripts
./create_test -xml_mach yellowstone -xml_compiler intel -mach yellowstone_intel -xml_category aux_clm \
-compare clm4_5_90 -baselineroot /glade/p/work/$USER/myclm_baselines
```

This will compare results of a new set of tests with the previous testing you did for "myclm4_5_90".

Step 4c -- Step-2 TRUE, but not Step-1 -- run with generate

This is the same as Step-3 above.

Step 4d -- Both Step-1 and Step-2 TRUE -- run with BOTH compare AND generate

```
cd scripts
./create_test -xml_mach yellowstone -xml_compiler intel -mach yellowstone_intel -xml_category aux_clm \
-compare clm4_5_90 -generate mynewclm4_5_90 -baselineroot /glade/p/work/$USER/myclm_baselines
```

Step 5 -- Examine Test Results

The test suite will automatically setup, build and then submit each test to the batch queue when it is executed. You want to wait until the tests have run in the batch queue before you check their status. But, after submitting the tests a script to check results is created called "cs.status.<jobid>.yellowstone" where jobid is the ID number given to the tests.

```
cd scripts
./cs.status.<jobid>.yellowstone
```

Example results of above command for a non-compare case (<jobid> in this case is 220217):

```
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220217
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220217.memleak
SFAIL ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-NoVSNoNI.220217
SFAIL ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-rootlit.220217
SFAIL SMS_D_Lm1_Mmpi-serial.CLM_USRDAT.I1PTCLM45.yellowstone_intel.220217
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220217
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220217.memleak
```

For each test you see a clearly defined test status followed by the exact testname. The testname is in the format: <test_type_with_options>.<resolution>.<compset>.<machine>_<compiler>[.<user_testmods_dir] where <user_testmods_dir> is optional, when it is used it is a relative path under "csm_utils/Testlistxml/testmods_dirs" with the "/" in the path changed to "-".

Example results of above command for a compare case (<jobid> in this case is 220218):

```
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220218
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220218.memleak
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220218.tputcomp
PASS  ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220218.nlcomp
SFAIL ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-NoVSNoNI.220218
SFAIL ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-rootlit.220218
SFAIL SMS_D_Lm1_Mmpi-serial.CLM_USRDAT.I1PTCLM45.yellowstone_intel.220218
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220218
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220218.memleak
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220218.tputcomp
PASS  SMS_Ld5.f19_g16.IRCP45CLM45BGC.yellowstone_intel.clm-decStart.220218.nlcomp
```

Step 5a -- Example test that PASS

In the above example the test "ERI_D.T31_g37.ICLM45.yellowstone_intel.clm-SNICARFRC.220218" gives a PASS for the main test. You also see that the "memleak", "tputcomp" and "nlcomp" tests PASS as well. This is for the "memory-leak", "throughput-comparison" and "namelist-comparison" tests.

Step 5b -- Example test that FAIL

Tests that pass all return a "PASS" status. Tests that fail return various error codes.

SFAIL -- Failure in scripts creation

GEN --- Tests generated, not yet run (this probably means you asked for the status too soon)

CFAIL -- Tests failed to compile.

RUN -- Tests are being run (or they died while running). If the tests are no longer running -- assume they died.

BFAIL -- Baseline fail

BFAIL1 -- Baseline fail

BFAIL2 -- Baseline fail

In the above example the test "ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-NoVSNoNI.220218" fails with a SFAIL which means it failed in scripts generation so there must be a problem in the construction of this test.

To find out exactly why a given test failed, go into the directory for that specific test and examine the TestStatus and TestStatus.log file, and other log files associated with the specific test. So for example...

cd ERS_D.f10_f10.ICLM45BGC.yellowstone_intel.clm-NoVSNoNI.220218

cat TestStatus.log

Details of using create_test

Here we will go into further details of using create_test for CLM testing.

Help on Testing Tools for Public Release Versions

This guide is for the very latest development version of CLM on the CLM trunk. For help on earlier versions see the CLM User's Guide and CESM User's Guide.

The CLM User's guide has a discussion on automated tools for testing CLM:

CLM1.2 User's Guide discussion on testing tools

CESM1.2 User's Guide for the CESM testing script is... CESM1.2 User's Guide on Testing

Basic operation:

The "create_test" script when run goes through a list of various tests that depend on the machine and compiler as well as testlist. When create_test is first invoked it creates all the different cases needed for the test list, and then it will start building the needed executable's for submission to the batch queue (the -nobuild command line option can be used to NOT build after creation). By default after building the tests will then be submitted to the batch queue (change this using the -noautosubmit command line option). To then build and/or submit you use the

./cs.submit.<jobid>.yellowstone_intel

script.

Test results are returned into the

./cs.status.<jobid>.yellowstone_intel

script. You have to wait until all the jobs are returned from the batch queue to see results. If you run the status script just after submission tests will show a status of "GEN".

Help on create_test:

Like other tools the "-help" option can be used to get a list of all the command line options and how to use them.

./create_test -help

To get a listing of the CLM specific test list use the "query_tests" script.

./query_tests -category aux_clm

You can also list by machine, compsets, and/or compilers.

Different Test lists:

The main test list useful for CLM testing is the "aux_clm" testlist. There is also an "aux_clm_short" testlist that has a very short fast testlist that is useful for debugging/development work. The "aux_clm" testlist is the list expected to be run as part of bringing CLM development to the trunk. There are also test lists that are specific for CESM development such as "prebeta". Note, that these test list will NOT work for a CLM tag -- only for a full CESM tag or checkout.

Running create_test for a single test case:

It's often useful to run create_test for a single test case. You may do this because you wonder if a specific test will pass because of the type of changes you've made. Or maybe in running a list of tests you have a specific test that fails and you wan to JUST run that one test.

To run a single test you use the "testname" option and give the following input...

<test_type_with_options>.<resolution>.<compset>.<machine>_<compiler>

./create_test -testname ERS_D.f10_f10.ICLM45BGC.yellowstone_intel

To give a test with namelist options use the "-user_testmods_dir" option to specify the directory with user_nl_* namelist files as well as xmlchange_cmds to change env settings. For example to specify the directory: csm_utils/Testlistxml/testmods_dirs/clm/NoVSNoNI do...

./create_test -testname ERS_D.f10_f10.ICLM45BGC.yellowstone_intel -user_testmods_dir csm_utils/Testlistxml/testmods_dirs/clm/NoVSNoNI

Note, if you have a specific set of namelist options and/or env settings that you want to invoke you could create your own directory to put user_nl_* files as well as xmlchange_cmds for env settings. This would be useful for testing new features that you are adding into the system.

Examining Test Results:

Test results are put into the "cs.status.<jobid>.yellowstone_intel" script that can then be run to see the status of each test. Note filter out tputcomp.

Namelist only comparison testing:

The "-nlcompareonly" command line option can be used to ONLY compare namelists for different versions. This is useful to track down differences in answers and find out it they are the result of changes in the code or changes in namelist options. To use this you will need to create baseline versions first and then run with the -nlcompareonly option to compare to your baseline.

```
cd <baselinedir>/scripts
./create_test -xml_category aux_clm -xml_mach yellowstone -xml_compiler intel -nlcompareonly -generate
mybaseline
cd <testdir>/scripts
./create_test -xml_category aux_clm -xml_mach yellowstone -xml_compiler intel -nlcompareonly -compare mybaseline
```

This type of testing is very fast, since it just creates cases and compares the namelists. It is useful when you are making namelist changes as well as other code changes and want to make sure your tests have identical namelists for example. Or to make sure the changes in namelists are what you expect them to be, and only change for the tests that you expect might change.

Why you wouldn't want to do this -- you already know all of your namelists are different anyway.

Why you WOULD want to do this -- to verify that your namelists are identical as you expect.

What to do when tests fail:

When you run the script "./cs.status.<jobid>.yellowstone_intel" it will report on the status for each test that was run.

Looking up expected fail status for a given CLM tag:

There is an XML file in the directory "models/lnd/clm/bld/unit_testers/xFail" called "expectedClmTestFails.xml" that lists the fails that are expected to fail for the given CLM tag. Tests are listed by test type (or for example list), machine and compiler. This list also gives the status of CLM Tools tests as well as CLM build-namelist tests.

Testing Best Practices

Here are recommended guidelines for working with the automated test scripts for your development. The two main points are to test often, but don't let it slow down your work too much, and separate your changes into ones that change answers and those that don't. You can validate that your changes that you don't think will change answers are correct, by using the test suite to ensure that they indeed do not.

Frequency

The purpose of testing is to make it easier to find problems with changes that you make. If you don't test often enough, you'll have a large set of changes and when you find problems in them with your testing -- it will be difficult to find and fix them. If you test too often (which is honestly difficult to do), the time spent testing will dwarf your development time. Debugging problems in the entire code base or even in a large set of changes can be extremely time-consuming and frustrating. Therefore frequent testing is a must.

Separate Development into answer changing and non-answer changing

One way to validate that a set of non-answer changing modifications are correct is to use the test suite to validate that it truly does NOT change answers. So it's useful to separate your answer changing modifications from

Checkin changes to your branch after you have a successful set of testing done (or after you've run long simulations that are successful)

Again the point of testing is to make it easier to find problems with code changes. So when you have a set of code that is working you'll want to check it into your branch so that you can then compare it to your next set of code changes that you are working on.

Use the single test option for development

To get quicker turnaround time for your testing in the midst of your development, use the single test option for a low resolution test. Once that is working go to using the longer suite of tests.

Use the single test option to return individual tests that fail from the longer test list

When you find isolated problems from the list of tests, it's usually easier to run the single test that fails rather than constantly running the entire test list.

In general ignore throughput tests

Because of variability in machines the throughput tests (that compare the total execution time of your code to the baseline) are NOT reliable enough to use. In general you shouldn't worry about needed science

CLM Tools Testing Page