Unit Testing in CESM

Introducing a new tool set

Sean Patrick Santos

National Center for Atmospheric Research

19th Annual CESM Workshop, 2014

Outline

Motivation

Motivation

- We have automated system tests already, which are:

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.
  - Often highly useful for developers, but too slow (>= 10 min!).

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.
  - Often highly useful for developers, but too slow (>= 10 min!).
  - Coarse. Tracking down a problem requires detailed reasoning or trial-and-error.

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.
  - Often highly useful for developers, but too slow (>= 10 min!).
  - Coarse. Tracking down a problem requires detailed reasoning or trial-and-error.
- A unit test framework simplifies writing and running tests of isolated subroutines.

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.
  - Often highly useful for developers, but too slow (>= 10 min!).
  - Coarse. Tracking down a problem requires detailed reasoning or trial-and-error.
- A unit test framework simplifies writing and running tests of isolated subroutines.
- Standard unit test suites preserve tests that developers might otherwise lose or abandon.

Motivation

- We have automated system tests already, which are:
  - Indispensible for regression testing.
  - Often highly useful for developers, but too slow (>= 10 min!).
  - Coarse. Tracking down a problem requires detailed reasoning or trial-and-error.
- A unit test framework simplifies writing and running tests of isolated subroutines.
- Standard unit test suites preserve tests that developers might otherwise lose or abandon.
- Fast, automated tests allow for agile development.

Current Tool Status

Current Tool Status

- A new script (`run_tests.py`) can run automated unit test suites for each component with one command.

Current Tool Status

- A new script (`run_tests.py`) can run automated unit test suites for each component with one command.
- After each set of code modifications, rebuilding and rerunning take seconds (or less).

Current Tool Status

- A new script (`run_tests.py`) can run automated unit test suites for each component with one command.
- After each set of code modifications, rebuilding and rerunning take seconds (or less).
- New scripts and CMake modules are present in CESM 1.3 beta tags, and planned for release next year.

Current Tool Status

- A new script (`run_tests.py`) can run automated unit test suites for each component with one command.
- After each set of code modifications, rebuilding and rerunning take seconds (or less).
- New scripts and CMake modules are present in CESM 1.3 beta tags, and planned for release next year.
- Unit test suites are already being used in development versions of CLM and csm_share.

Current Tool Status

- A new script (`run_tests.py`) can run automated unit test suites for each component with one command.
- After each set of code modifications, rebuilding and rerunning take seconds (or less).
- New scripts and CMake modules are present in CESM 1.3 beta tags, and planned for release next year.
- Unit test suites are already being used in development versions of CLM and csm_share.
  - CAM unit tests planned for trunk commit in July.

External Tools

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.
  - Provides a generic Fortran test driver.

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.
  - Provides a generic Fortran test driver.
- CMake/CTest

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.
  - Provides a generic Fortran test driver.
- CMake/CTest
  - Developed by Kitware, funded by the NIH.

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.
  - Provides a generic Fortran test driver.
- CMake/CTest
  - Developed by Kitware, funded by the NIH.
  - CMake builds subsets of the source code.

External Tools

- pFUnit (http://sourceforge.net/p/pfunit)
  - "xUnit" style test framework for parallel Fortran 2003, developed at NASA (Goddard Space Flight Center).
  - Preprocessor and utility routines simplify adding tests and improve the output for failed tests.
  - Provides a generic Fortran test driver.
- CMake/CTest
  - Developed by Kitware, funded by the NIH.
  - CMake builds subsets of the source code.
  - CTest aggregates tests from multiple executables.

CSEG Development

CSEG Development

- `run_tests.py`

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.
  - Integrates with CESM's `Machines/` files.

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.
  - Integrates with CESM's `Machines/` files.
- CMake_Fortran_utils

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.
  - Integrates with CESM's `Machines/` files.
- CMake_Fortran_utils
  - Provides utility functions for CESM builds (e.g. the genf90.pl preprocessor).

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.
  - Integrates with CESM's `Machines/` files.
- CMake_Fortran_utils
  - Provides utility functions for CESM builds (e.g. the genf90.pl preprocessor).
  - Contains hooks for CESM `Machines/` information.

CSEG Development

- `run_tests.py`
  - CESM-specific Python script.
  - Simplifies building and running one or more CTest suites out of source.
  - Integrates with CESM's `Machines/` files.
- CMake_Fortran_utils
  - Provides utility functions for CESM builds (e.g. the genf90.pl preprocessor).
  - Contains hooks for CESM `Machines/` information.
  - Provides functions to handle pFUnit build and test output.

Outline

Overview

Top Level CMakeLists.txt

CLM source listing

```
|> ls -1 ${CESMROOT}/models/lnd/clm/src/
clm4_0
clm4_5
CMakeLists.txt
cpl
README.unit_testing
unit_test_mocks
unit_test_shr
util_share
```

Top Level CMakeLists.txt

CLM source listing

```
|> ls -1 ${CESMROOT}/models/lnd/clm/src/
clm4_0
clm4_5
CMakeLists.txt
cpl
README.unit_testing
unit_test_mocks
unit_test_shr
util_share
```

Using run_tests.py

Example with CTest output

```
|> cd ${CESMROOT}/tools/unit_testing
|> run_tests.py --compiler=nag \
|>    --test-spec-dir=${CESMROOT}/models/lnd/clm/src \
|>    --build-dir=${TEMPDIR}/clm_tests
   <<<Tons of CMake/CTest output.>>>

100% tests passed, 0 tests failed out of 7

Total Test time (real) =   0.15 sec
```

Test Failures

Example with a CTest failure

```
|> export CTEST_OUTPUT_ON_FAILURE=TRUE
|> run_tests.py --compiler=nag \
|>     --test-spec-dir=${CESMROOT}/models/lnd/clm/src \
|>     --build-dir=${TEMPDIR}/clm_tests
    <<<Tons of CMake/CTest output.>>>
1/7 Test #1: daylength .........................***Failed
Error regular expression found in output.
Regex=[FAILURES!!!]  0.00 sec
    <<<pFUnit and CTest output.>>>
86% tests passed, 1 tests failed out of 7

Total Test time (real) =   0.08 sec

The following tests FAILED:
  1 - daylength (Failed)
Errors while running CTest
```

pFUnit Output

pFUnit output when "test_near_poles" fails

```
..F.....
Time:           0.001 seconds

 Failure in: test_near_poles
   Location: [test_daylength.pf:31]
expected: +0.000000 but found: +1.000000;
difference: |+1.000000| > tolerance:+0.1000000E-02;
first difference at element [1].

 FAILURES!!!
Tests run: 7, Failures: 1, Errors: 0
```

Overview

A Short pFUnit Example

test_daylength.pf (excerpt)

```
@Test
subroutine test_near_poles()
   ! Tests points near the north and south
   ! pole, which should result in full night
   ! and full day
   @assertEqual([0.0_r8, 86400.0_r8],
     daylength([-1.5_r8, 1.5_r8], 0.1_r8),
     tolerance=tol)
end subroutine test_near_poles
```

A Short pFUnit Example

- `@` marks pFUnit directives for the preprocessor.

test_daylength.pf (excerpt)

```
@Test
subroutine test_near_poles()
   ! Tests points near the north and south
   ! pole, which should result in full night
   ! and full day
   @assertEqual([0.0_r8, 86400.0_r8],
     daylength([-1.5_r8, 1.5_r8], 0.1_r8),
     tolerance=tol)
end subroutine test_near_poles
```

A Short pFUnit Example

- `@` marks pFUnit directives for the preprocessor.
- Otherwise, a normal module using `pfunit_mod`.

test_daylength.pf (excerpt)

```
@Test
subroutine test_near_poles()
   ! Tests points near the north and south
   ! pole, which should result in full night
   ! and full day
   @assertEqual([0.0_r8, 86400.0_r8],
     daylength([-1.5_r8, 1.5_r8], 0.1_r8),
     tolerance=tol)
end subroutine test_near_poles
```

pFUnit Output

pFUnit output when "test_near_poles" fails

```
..F.....
Time:           0.001 seconds

 Failure in: test_near_poles
   Location: [test_daylength.pf:31]
expected: +0.000000 but found: +1.000000;
difference: |+1.000000| > tolerance:+0.1000000E-02;
first difference at element [1].

 FAILURES!!!
Tests run: 7, Failures: 1, Errors: 0
```

Workflow Summary - pFUnit Tests

Workflow Summary - pFUnit Tests

1. Make a module that uses `pfunit_mod`.

Workflow Summary - pFUnit Tests

1. Make a module that uses `pfunit_mod`.
2. Write a test subroutine that uses one of the `@assert` directives to test a condition.

Workflow Summary - pFUnit Tests

1. Make a module that uses `pfunit_mod`.
2. Write a test subroutine that uses one of the `@assert` directives to test a condition.
3. Annotate each routine with `@Test`.

Workflow Summary - pFUnit Tests

1. Make a module that uses `pfunit_mod`.
2. Write a test subroutine that uses one of the `@assert` directives to test a condition.
3. Annotate each routine with `@Test`.
4. Additional features, such as test fixtures, reduce duplication between tests (see appendix).

Non-pFUnit Tests

Non-pFUnit Tests

- pFUnit is generally more convenient, but does not support PGI or older versions of GNU.

Non-pFUnit Tests

- pFUnit is generally more convenient, but does not support PGI or older versions of GNU.
- Have not yet selected any specific frameworks for non-Fortran code.

Non-pFUnit Tests

- pFUnit is generally more convenient, but does not support PGI or older versions of GNU.
- Have not yet selected any specific frameworks for non-Fortran code.
- If not using pFUnit, you are on your own, but any test program compatible with CTest should work.

Overview

Source Directory CMakeLists.txt

CMakeLists.txt (CLM biogeophys)

```
# Note that this is just used for unit
# testing; hence, we only need to add source
# files that are currently used in unit tests

list(APPEND clm_sources DaylengthMod.F90)

sourcelist_to_parent(clm_sources)
```

Source Directory CMakeLists.txt

- Add any new sources to the component's source list.

CMakeLists.txt (CLM biogeophys)

```
# Note that this is just used for unit
# testing; hence, we only need to add source
# files that are currently used in unit tests

list(APPEND clm_sources DaylengthMod.F90)

sourcelist_to_parent(clm_sources)
```

Source Directory CMakeLists.txt

- Add any new sources to the component's source list.
- `sourcelist_to_parent` exports the source list.

CMakeLists.txt (CLM biogeophys)

```
# Note that this is just used for unit
# testing; hence, we only need to add source
# files that are currently used in unit tests

list(APPEND clm_sources DaylengthMod.F90)

sourcelist_to_parent(clm_sources)
```

Test Directory CMakeLists.txt

CMakeLists.txt (linear_1d_operators)

```
create_pFUnit_test(daylength test_daylength_exe
  "test_daylength.pf" "")

target_link_libraries(test_daylength_exe clm
  csm_share)
```

Test Directory CMakeLists.txt

- `create_pFUnit_test` accepts a test name, executable name, pFUnit sources, and Fortran sources, and adds a pFUnit executable to CTest.

CMakeLists.txt (linear_1d_operators)

```
create_pFUnit_test(daylength test_daylength_exe
  "test_daylength.pf" "")

target_link_libraries(test_daylength_exe clm
  csm_share)
```

Test Directory CMakeLists.txt

- `create_pFUnit_test` accepts a test name, executable name, pFUnit sources, and Fortran sources, and adds a pFUnit executable to CTest.
- `target_link_libraries` used to add libraries for CLM. Due to differences in build-time options, CAM will use a source-list based method instead (see appendix).

CMakeLists.txt (linear_1d_operators)

```
create_pFUnit_test(daylength test_daylength_exe
  "test_daylength.pf" "")

target_link_libraries(test_daylength_exe clm
  csm_share)
```

Workflow Summary

Workflow Summary

1. Add CESM sources to their directories' CMakeLists.txt.

Workflow Summary

1. Add CESM sources to their directories' CMakeLists.txt.
2. Create a test subdirectory and call `add_subdirectory` on it in the top level CMakeLists.txt file.

Workflow Summary

1. Add CESM sources to their directories' CMakeLists.txt.
2. Create a test subdirectory and call `add_subdirectory` on it in the top level CMakeLists.txt file.
3. In CMakeLists.txt for your unit test subdirectory, add an executable and a test.

Workflow Summary

1. Add CESM sources to their directories' CMakeLists.txt.
2. Create a test subdirectory and call `add_subdirectory` on it in the top level CMakeLists.txt file.
3. In CMakeLists.txt for your unit test subdirectory, add an executable and a test.
4. Link to (or directly add) the CESM source code you are testing.

Outline

To-Do List

To-Do List

- Add more unit tests!

To-Do List

- Add more unit tests!
- Come up with a strategy for leveraging information about batch systems in `Machines/` to run MPI tests on clusters.

To-Do List

- Add more unit tests!
- Come up with a strategy for leveraging information about batch systems in `Machines/` to run MPI tests on clusters.
- Improve `run_tests.py` output.

To-Do List

- Add more unit tests!
- Come up with a strategy for leveraging information about batch systems in `Machines/` to run MPI tests on clusters.
- Improve `run_tests.py` output.
  - Divert CMake output to log file.

To-Do List

- Add more unit tests!
- Come up with a strategy for leveraging information about batch systems in `Machines/` to run MPI tests on clusters.
- Improve `run_tests.py` output.
  - Divert CMake output to log file.
  - Leverage recent pFUnit improvements (e.g. verbose and XML outputs).

Acknowledgements

- Tom Clune and Michael Rilee, for regularly updating pFUnit in response to our feedback.
- Bill Sacks, for being an early adopter, proponent, and contributor.
- Everyone who gave early feedback and/or were early adopters (especially CLM developers).

NCAR
NATIONAL CENTER FOR ATMOSPHERIC RESEARCH

NSF

U.S. DEPARTMENT OF ENERGY
Office of Science

Resources

- Martin Fowler on the definition of "unit test": http://martinfowler.com/bliki/UnitTest.html
- The pFUnit home page: http://sourceforge.net/p/pfunit
- CESM and PIO CMake modules: https://github.com/CESM-Development/CMake_Fortran_utils
- This presentation (once posted): http://www.cesm.ucar.edu/events/workshops.html

Other Features

- Only changed files are rebuilt unless `-clean` is passed.
- Using CESM or CESM_DEBUG as the build type will extract the corresponding compiler flags from Machines. (Set by `-build-type`.)
- `-ctest-args` and `-verbose` can be used to change flags sent to the underlying commands.

pFUnit test fixtures

Global data Annotate setup and teardown routines with `@Before` and `@After`, respectively.

Test-specific data Subclass `TestClass` with a type that has the data you need, and annotate it with `@TestCase`. The `setUp`, `tearDown`, and test routines will be passed an argument of your type called `this`.

test_fd_solver.pf

Module header

```
module test_fd_solver
use pfunit_mod
! <More use statements, implicit none, comments>

! Grid size used by these tests.
integer, parameter :: n = 101

! The grid itself (mid-points and distances
! between points).
real(r8) :: x(1,n), deltas(1,n-1)

contains
! <Continued...>
```

test_fd_solver.pf

setUp/tearDown

```
@Before
subroutine setUp()
  integer :: i
  ! Grid is n points between 0 and 1.
  x(1,:) = [( real(i, r8) / real(n-1, r8), i = 0, n-1 )]
  ! Introduce nonuniformity.
  x = x*x
  deltas = x(:,2:) - x(:,:n-1)
end subroutine setUp

@After
subroutine tearDown()
  ! Fight pollution!
  x = 0._r8
  deltas = 0._r8
end subroutine tearDown
```

test_fd_solver.pf

solves_decay

```
@Test
subroutine solves_decay()
  ! Time step.
  real(r8) :: dt
  ! PDE coefficients.
  real(r8) :: coef_q(1,n)
  ! Array to evolve.
  real(r8) :: q(1,n), q_expected(1,n)
  ! Decomposed diffusion matrix.
  type(lu_decomp) :: diff_decomp

  ! Equation to solve is dq/dt = -q
  coef_q = -1._r8
  dt = 1._r8
  ! Decomposition
  diff_decomp = vd_lu_decomp(dt, deltas, &
    coef_q=coef_q)
```

solves_decay

```
  ! We are seeking the solution
  ! q(x,t) = e^(-t) * cos(pi*x)
  ! Set q for t = 0.
  q = cos(pi*x)
  ! Expected result after one step.
  q_expected = cos(pi*x)*exp(-dt)

  call diff_decomp%solve(q)

  ! Max error in this case is
  ! (1/2 - 1/e)*dt*maxval(x)
  ! which is
  ! ~dt*maxval(x)/6.
  @assertEqual(q_expected, q, &
    tolerance=dt*maxval(x)/6._r8)
end subroutine solves_decay
```

Suggested Workflow Summary - Non-pFUnit Tests

1. Define (or borrow) a very basic `assert` subroutine that can signal a test failure to CMake (e.g. using a non-zero return code).
2. Write a minimal program for each test.
3. Make separate modules for shared setup/teardown routines.

A Short Non-pFUnit Example

- `assert` is defined off-screen, using `stop 1`.
- Conundrum: a dozen executables, or a dozen asserts in just one?

test_infnan.F90 (excerpt)

```
real(r8) :: inf
integer(i8), parameter :: dpinfpat = &
   int(O'07776000000000000000000',i8)
inf = transfer(dpinfpat,inf)

call assert(shr_infnan_isposinf( inf ), &
   "Test that value set to inf is inf" )
```

Top Level CMakeLists.txt (Approach 1)

- Include the `CESM_utils` module.
- First subdirectories define global sourcelist variables.
- Further subdirectories contain unit tests.

CMakeLists.txt (CAM, excerpt)

```
list(APPEND CMAKE_MODULE_PATH ${CESM_CMAKE_MODULE_DIRECTORY})
include(CESM_utils)

set(CAMROOT ../../)
set(CESMROOT ${CAMROOT}../../../)
add_subdirectory("${CESMROOT}/models/csm_share/shr" csm_share)
list(APPEND cam_sources ${share_sources})
add_subdirectory(${CAMROOT}src/physics/cam physics_cam)

add_subdirectory(linear_1d_operators)
add_subdirectory(vdiff_lu_solver)
```

Test Directory CMakeLists.txt (Approach 1)

- `extract_sources` expands basenames to absolute paths.
- `create_pFUnit_test` handles pFUnit builds (including preprocessing), and adds the test to CTest.

CMakeLists.txt (linear_1d_operators)

```
# Local pFUnit files.
set(pf_sources test_diagonal.pf test_derivatives.pf
  test_arithmetic.pf)

# Sources to test.
set(sources_needed shr_kind_mod.F90 linear_1d_operators.F90)
extract_sources("${sources_needed}" "${cam_sources}"
  test_sources)

# Do source preprocessing and add the executable.
create_pFUnit_test(linear_1d_operators linear_1d_operators_exe
  "${pf_sources}" "${test_sources}")
```

Top Level CMakeLists.txt (Approach 2)

- Similar to Approach 1, except that `${clm_sources}` is used to create a library before adding test directories.

CMakeLists.txt (CLM, excerpt)

```
add_library(csm_share ${share_sources})
add_library(clm ${clm_sources})
add_dependencies(clm csm_share)
```

Test Directory CMakeLists.txt (Approach 2)

- Using `extract_sources` is not necessary.
- Instead, link against libraries that are already added.

CMakeLists.txt (excerpt from CLM)

```
set(pfunit_sources
  test_update_landunit_weights_one_gcell.pf
  test_update_landunit_weights.pf)

create_pFUnit_test(dynLandunitArea
  test_dynLandunitArea_exe "${pfunit_sources}" "")

target_link_libraries(test_dynLandunitArea_exe clm
  csm_share)
```

Limitations

- pFUnit has more limited compiler support than CESM.
  - PGI's Fortran 2003 support is not adequate to compile pFUnit, largely due to compiler bugs.
  - You need a very recent version of GCC to compile with gfortran (4.8 for pFUnit 2.1, 4.9 for pFUnit 3.0).
- Making CTest work on MPI code on some HPC systems is not trivial.

Missing Features

- The CMake/CTest system currently doesn't help you much with batch systems.
- There's no connection between aggregation at different levels. CTest treats pFUnit executables as a single test (but can print pFUnit's output, so you can still see how many tests failed).
- CTest is only pass/fail; there's no recognized way to skip tests.
- Not all pFUnit capabilities are leveraged. (E.g. there's a verbose output mode that we don't provide a way to turn on via CTest.)